

OsborneMcGraw-Hill

Macintosh™

GRAPHICS AND SOUND

55055 1285
12 \$34.95



PROGRAMMING IN MICROSOFT® BASIC

DAVID A. KATER

MacintoshTM Graphics and Sound

Programming in Microsoft® BASIC

David A. Kater

Osborne McGraw-Hill
Berkeley, California

Disclaimer of Warranties and Limitation of Liabilities

The authors have taken due care in preparing this book and the programs in it, including research, development, and testing to ascertain their effectiveness. The authors and the publisher make no expressed or implied warranty of any kind with regard to these programs nor the supplementary documentation in this book. In no event shall the authors or the publishers be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance, or use of any of these programs.

COPYRIGHT. This collection of programs and their documentation is copyrighted. You may not copy or otherwise reproduce any part of any program in this collection or its documentation, except that you may load the programs into a computer as an essential step in executing the program on the computer. You may not transfer any part of any program in this collection electronically from one computer to another over a network. You may not distribute copies of any program or its documentation to others. Neither any program nor its documentation may be modified or translated without written permission from Osborne/McGraw-Hill.

NO WARRANTY OF PERFORMANCE. Osborne/McGraw-Hill does not and cannot warrant the performance or results that may be obtained by using any program in this book. Accordingly, the programs in this collection and their documentation are sold "as is" without warranty as to their performance, merchantability, or fitness for any particular purpose. The entire risk as to the results and performance of each program in the collection is assumed by you. Should any program in this collection prove defective, you (and not Osborne/McGraw-Hill or its dealer) assume the entire cost of all necessary servicing, repair, or correction.

LIMITATION OF LIABILITY. Neither Osborne/McGraw-Hill nor anyone else who has been involved in the creation, production, or delivery of these programs shall be liable for any direct, incidental, or consequential benefits, such as, but not limited to, loss of anticipated profits or benefits, resulting from the use of any program in this collection or arising out of any breach of any warranty. Some states do allow the exclusion or limitation of direct incidental or consequential damages, so the above limitation may not apply to you.

Published by
Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne McGraw-Hill at the above address.

Macintosh is a trademark of Apple Computer, Inc.
Microsoft is a registered trademark of Microsoft Corporation.
A complete list of trademarks appears on page 271.

Macintosh™ Graphics and Sound

Copyright ©1986 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DODO 898765

ISBN 0-07-881177-5

Jonathan Erickson, Acquisitions Editor
Kay Luthin, Copy Editor
Jan Benes, Text Design

Deborah Wilson, Composition
Yashi Okita, Cover Design
Kay Nelson, Technical Editor

Contents

	<i>Introduction</i>	vii
Chapter 1	<i>Macintosh Graphics</i>	1
Chapter 2	<i>Printing and Plotting</i>	13
Chapter 3	<i>Drawing Basic Shapes and Patterns</i>	51
Chapter 4	<i>Interactive Graphics</i>	93
Chapter 5	<i>Sound</i>	137
Chapter 6	<i>Transferring Images</i>	167
Chapter 7	<i>Animation Techniques</i>	199
Chapter 8	<i>Manipulating Displays and Viewing Objects</i>	223
Chapter 9	<i>Designing Efficient Programs</i>	259
	<i>Index</i>	273

Dedication

To Mom
Who is about as graphic as they come

Acknowledgments

I have especially enjoyed writing this book because it has given me the opportunity to work with a lot of talented people. Thanks to everyone who contributed their time, energy, and creativity.

First and foremost, I'd like to thank Renda Ozden for his dedication, creative insight, graphics and programming talent, and companionship throughout this project. Renda designed and implemented many of the programs, drew several graphics illustrations, and was an endless source of ideas for graphics applications. This book is very much a product of his fertile imagination.

Russel Schnapp, author of *Macintosh Graphics in Modula-2* (Prentice-Hall, 1986), gave significant help at a critical time.

Nick Galemo, author of the popular Paint Mover program, edited and revised the chapters on sound and on working with screen images. His timely contribution helped to keep the book on schedule.

Anthony Mack, President of the UC Irvine Mac Users Group, thoroughly tested each of the program listings in the book, making modifications as needed to give the programs a uniform format. He also drew several illustrations.

Morgan Davis contributed several programs, including Towers of Hanoi and the cursor editor.

Dick Kater, my co-author in *Getting the Most Out of Your Epson Printer* (McGraw-Hill, 1985) and *The Printed Word* (Microsoft Press, 1985), made several helpful suggestions.

The EduKater staff provided support and encouragement throughout the project. Ramona Garcia managed the office and was tenacious about getting the chapters out on time. Joe Perez took time off from accounting to write several programs. Griselda Engelhorn, Maureen Aldrich, and Tina Vitous kept everyone's spirits high and did whatever was necessary to keep the project going.

Introduction

No matter how you look at it, learning about computers and how to program them takes time; and the more complex the subject, the more draining it can be, except when you have a particular interest in what you are learning. Motivation makes the difference. Fortunately, it is easy to get excited about computer graphics and sound, since both appeal to our creative nature.

In this book, you have an opportunity to learn about Macintosh graphics and sound by typing and running short programs. You'll find that actively using the programs in the book is the best way to learn how to use graphics statements, so try out the programs. Read the annotated program listings, paying close attention to the ways various statements and functions are used.

Above all, don't expect to learn all about computer graphics in one sitting. Be patient, and you will be rewarded with newfound graphics abilities.

Using This Book

Macintosh Graphics and Sound is designed to usher you into the world of Macintosh graphics and sound by using BASIC. BASIC (short for Beginner's All-purpose Symbolic Instruction Code) is a programming language for beginners. We will develop this book around the first BASIC introduced on the Mac: version 2.0 of Microsoft BASIC.

The challenge for any writer of computer subjects is to bridge the gap between a tutorial and a reference work. A pure reference work is often only marginally useful to the computer novice; a long, drawn-out tutorial is the last thing a polished programmer wants to wade through. *Macintosh Graphics and Sound* contains a combination of both approaches. The book is organized around graphics concepts. It starts with such basic topics as working with the video display and drawing simple shapes, and then progresses to more advanced topics like rotating two- and three-dimensional objects. It is well indexed so that you can find your way quickly to topics of interest.

For those who prefer a 'learn by doing' approach, graphics statements and techniques are illustrated with short sample programs. By reading and experimenting with the programs, you can discover what the Macintosh has to offer. Each chapter concludes with several comprehensive programs that apply the concepts and techniques learned in that chapter.

System Requirements

To get the most out of this book, your minimal hardware requirement is the original Macintosh, which has 128K of memory. You can write short programs, but you will have to be very conscious of memory limitations. Most of the programs in this book work on the 128K Mac; exceptions are carefully noted. For those who have 128K machines, Chapter 9 discusses ways to deal with limited memory.

While a dual-drive system is necessary for word processing and some other applications, BASIC programmers can use a single-drive machine. With careful file management, you can store your operating system and BASIC, and still have nearly 200K for program storage.

A printer is not an absolute necessity for using BASIC, but it can make life a lot easier. Printouts make your work on the computer somewhat transportable. A printer also allows you to distribute paper copies of your graphics displays.

You may also wish to borrow or purchase a digitizer unit that converts real-life images into digitized form in the Mac.

For software, you'll need a copy of Microsoft BASIC. The programs in this book are written in version 2.0 of Microsoft BASIC. You'll occasionally need a graphics program, such as MacPaint or MacDraw, for creating graphics images.

READER BACKGROUND

This book is designed for all BASIC programmers who want to master Macintosh's remarkable graphics abilities. On other computers, the subject of graphics is generally avoided by beginning programmers. But the Mac invites even the beginner to participate in the graphics experience.

The first few chapters start at a leisurely pace so that the novice programmer can get started with ease. At each stage we develop the tools to prepare the reader for the next section. The book is also carefully indexed so that advanced programmers will be able to find quickly the sections they want to know about.

Newcomers to BASIC and the Mac will find it best to read the chapters in sequence. Take your time and read carefully. You'll find that running each of the sample exercises will help you learn the material in a surprisingly short time and to retain it longer. Working with practice programs lets you get all the catastrophic errors out of your system before you attempt your own programs. Those who wish to minimize typing time in working with the practice programs might consider purchasing the optional program disk.

If you are an experienced programmer, you will probably want to proceed at a faster pace. You may want to skim the first few chapters to get familiar with the Mac, and then use the Index and Table of Contents to find topics of interest. The applications programs at the end of each chapter provide examples of how to use the tools introduced in that chapter. The optional program disk will allow you to modify the major programs.

The Macintosh is a machine that allows us to explore our own creativity. We hope this book will inspire you to push your limits and to make new discoveries about yourself.

Program Disk Offer

We have made many of the major programs in this book available on disk. The files are stored on a data disk, so single-drive users may want to transfer the files to a system disk.

x Macintosh Graphics and Sound

The cost is \$19.95, which includes shipping to anywhere in the continental United States. Residents in other parts of the world, please add \$3.00 to cover additional shipping costs (\$22.95 total). California residents, please add 6% for state tax (\$21.15 total).

Make your check or money order payable to **EduKater**, and send it to:

David A. Kater
EduKater
P.O. Box 1868
La Mesa, CA 92041

Orders paid by money order or cashier's check will be shipped within three working days of receipt. Orders paid by personal check will be shipped within three weeks of receipt.

PROGRAM DISK OFFER

Enter quantity desired:

_____ diskette(s) at \$19.95: _____

Shipping outside U.S. (\$3.00): _____

Tax for CA residents (\$1.20): _____

Total enclosed: _____

Ship to:

1

Macintosh Graphics

Much has been written about how the Apple Macintosh computer sets a new standard for personal computing. One of the most intriguing innovations of the Macintosh is its easy-to-use graphics. This book can teach you how to use BASIC programs that will help you take full advantage of your Macintosh's graphics capabilities.

This chapter will give you an overview of what you can do with the BASIC programs in this book. A sample programming session will help you get started.

Macintosh Graphics and You

You may wonder why you would want to use BASIC to develop graphics on the Macintosh in the first place. After all, MacPaint,

2 Macintosh Graphics and Sound

MacDraw, Microsoft Chart, and other graphics-oriented software programs are available to satisfy the appetite of the most ardent graphics enthusiast. Each of these programs can give you useful, creative results. So why bother learning to program graphics when you can just *do* graphics?

The following examples answer this question. They illustrate some of the many ways BASIC programs can help you create and manage your graphics on the Mac.

BASIC AS A GRAPHICS MANAGER

BASIC can help you combine the graphics you have developed with other programs into complete presentations. You will learn in Chapter 6 how to transfer graphics images from other programs into BASIC. These images can be stored in memory or on disk and then

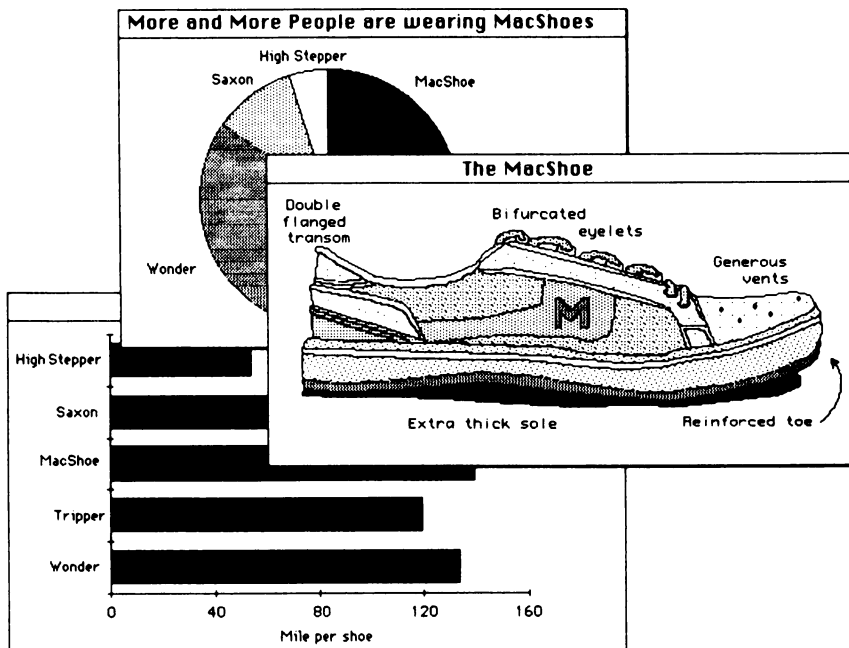


Figure 1-1.
BASIC can display frames created by Macintosh applications

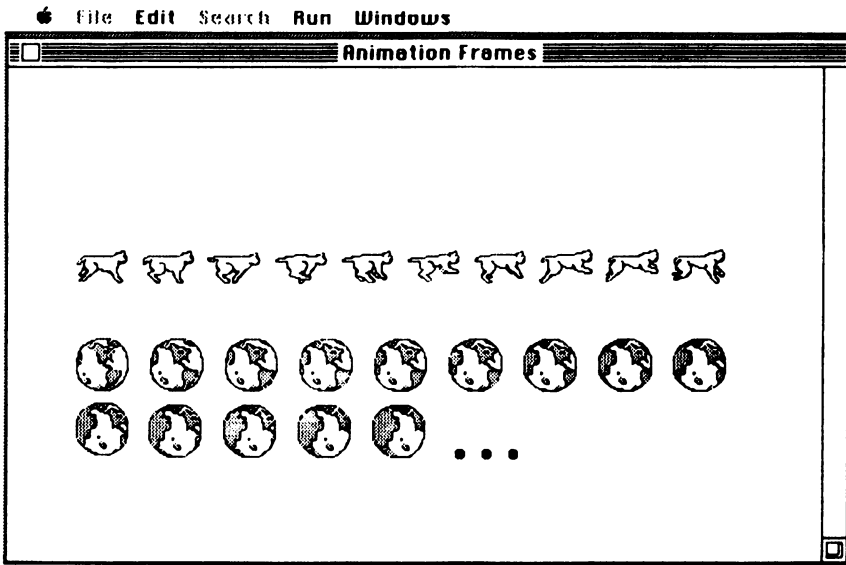


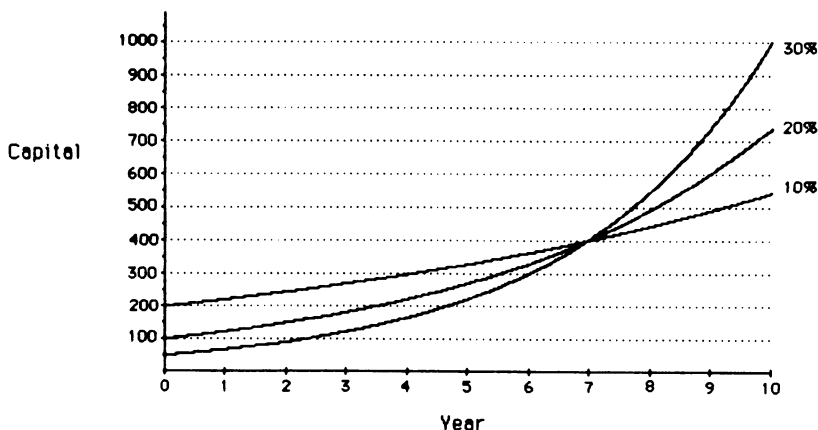
Figure 1-2.
Animation frames

displayed by a BASIC program as needed. For example, a retail sales outlet might use a BASIC program to manage a continuous slide-show sales floor demonstration, using charts from Chart, text from MacWrite, and illustrations from MacPaint. Figure 1-1 shows screens from a sample demonstration.

BASIC's abilities as a graphics manager don't stop with slide presentations. Macintosh BASIC provides several ways to store images in memory and then display them one at a time in succession to produce very believable animation. The images can be created with BASIC or some other program. Figure 1-2 shows two frame sequences created with a program developed in Chapter 7. One shows a cat leaping; the other shows the earth rotating on its axis.

PRECISION PLOTTING

BASIC also allows you to calculate and plot functions, like the one in Figure 1-3, quickly and efficiently. You can plot curves like those in Figure 1-3 with MacPaint or MacDraw, but it's much easier and more precise to plot them with BASIC. Figure 1-4 shows the short program segment necessary to plot the three curves.



```

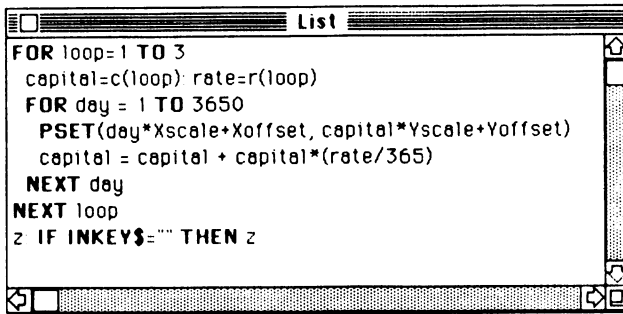
Xoffset = 70: Yoffset=240: Xscale=.1: Yscale = -.2
FOR i=1 TO 3
  READ c(i),r(i)
  DATA 50,30,100,20,200,10
NEXT i
FOR loop=1 TO 3
  capital=c(loop): rate=r(loop)
  FOR day = 1 TO 3650
    PSET(day*Xscale+Xoffset, capital*Yscale+Yoffset)
    capital = capital + capital*(rate/365)
  NEXT day
NEXT loop
Z: IF INKEY$="" THEN Z

```

Figure 1-3.
Function plot

PATTERNS AND ART

Part of the Macintosh's charm is the ease with which it can produce both serious business graphics and art for art's sake. MacPaint is an excellent tool for freehand drawing; BASIC is better suited for creating repetitive patterns and geometric shapes. The striking patterns shown in Figure 1-5 are created by a randomly repeating BASIC program that draws ovals gradually changing sizes and positions. A totally different effect is created by drawing lines between two curves, as shown in Figure 1-6.



```
FOR loop=1 TO 3
  capital=c(loop): rate=r(loop)
  FOR day = 1 TO 3650
    PSET(day*Xscale+Xoffset, capital*Yscale+Yoffset)
    capital = capital + capital*(rate/365)
  NEXT day
NEXT loop
Z IF INKEY$="" THEN Z
```

Figure 1-4.
Program code for growth curves

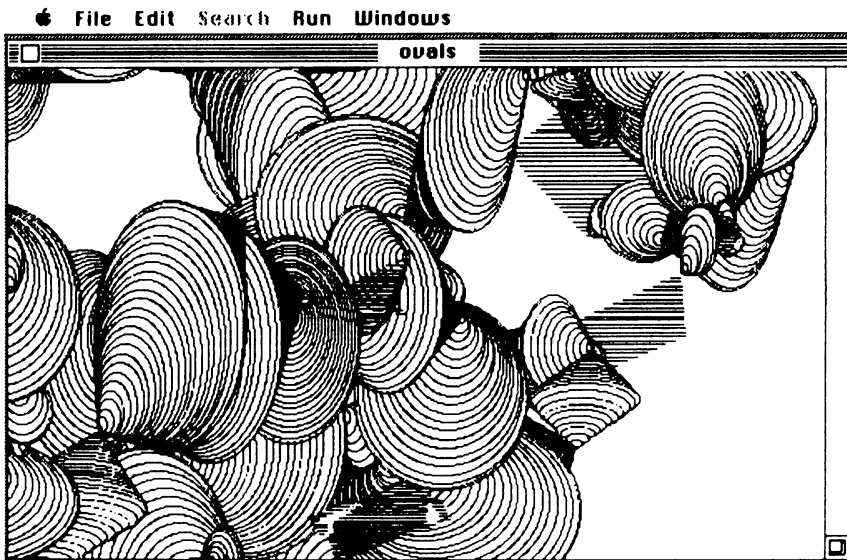


Figure 1-5.
Repeating ovals

INTERACTIVE GRAPHICS

One advantage of computer graphics over other forms of graphics is that the user can interact with the graphics images on a computer. Because BASIC can easily detect input from the mouse, you can write

6 Macintosh Graphics and Sound

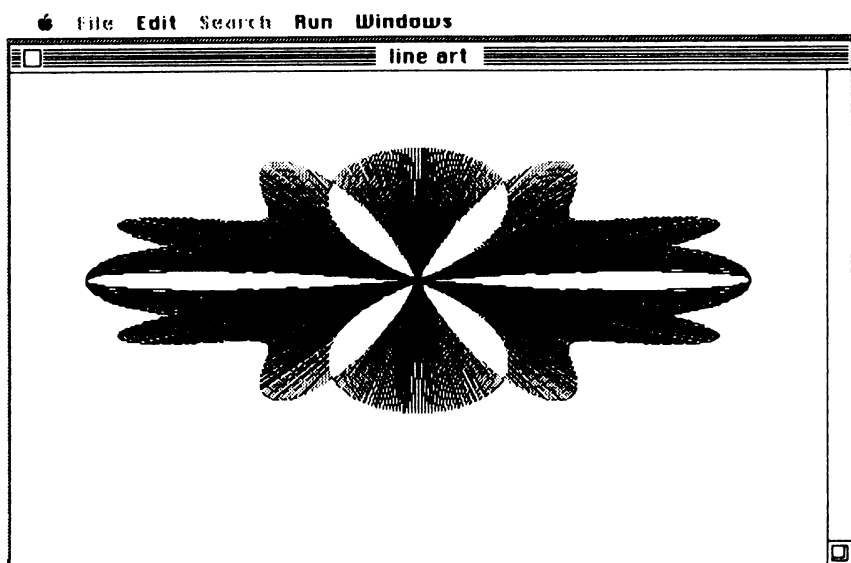


Figure 1-6.
Repeating line segments

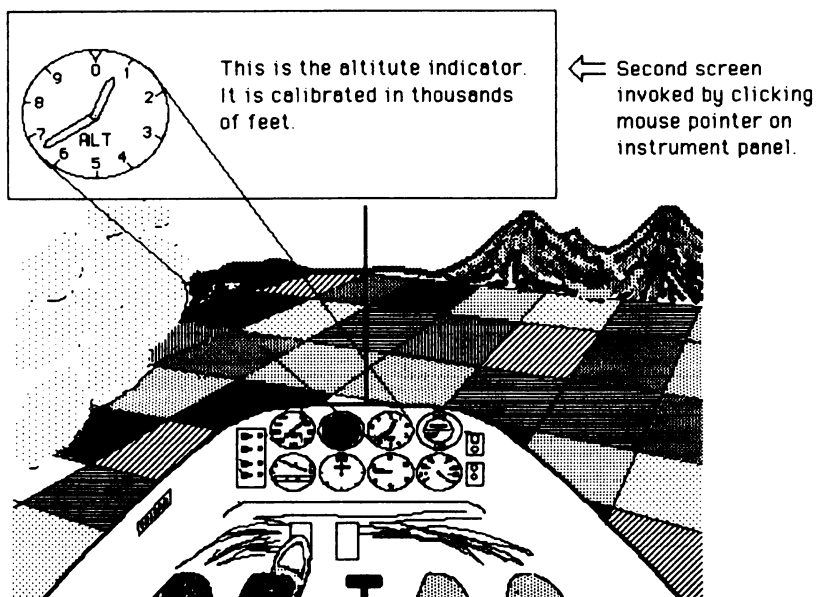


Figure 1-7.
Aircraft instrument panel

programs that will interact with the operator. This interactive ability can be used to full advantage in educational programs. Figure 1-7 illustrates how a tutorial program could introduce the viewer to the instrument panel of a small aircraft.

In this example you click the mouse pointer on any of the items on the panel. The program will respond with a close-up view of the item along with a description. This approach lets you control what you learn in a very intuitive and visual way. BASIC's ability to detect a mouse click anywhere on the screen is introduced in Chapter 4 and is used throughout the rest of the book.

The next example literally adds another dimension to the use of BASIC's interactive abilities. In Chapter 8, we introduce a program that allows you to draw three-dimensional schematic objects. Lines are drawn in panels representing three different views of the object. Figure 1-8 shows a schematic representation of a car. Once the object is drawn, the program allows you to view it from any direction.

While it is possible to draw three-dimensional objects with Mac-

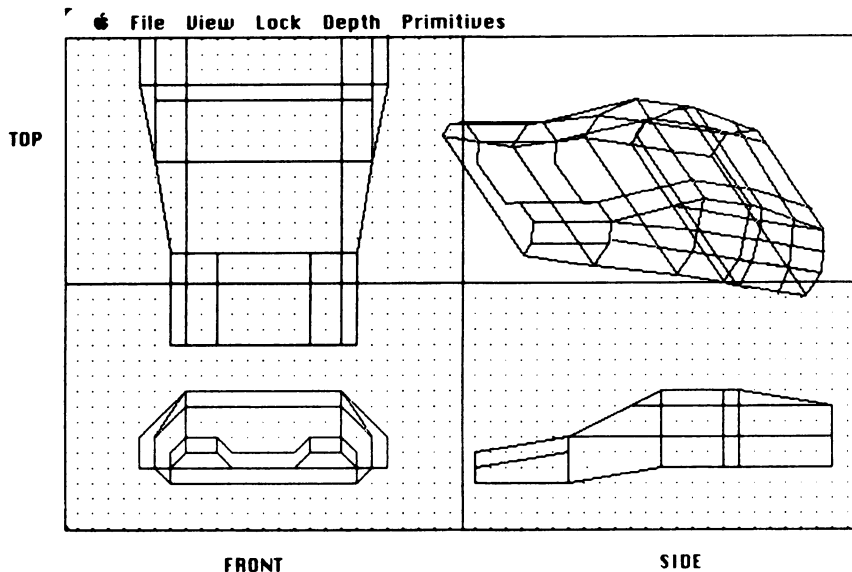


Figure 1-8.
Three-dimensional schematic depiction of car

Paint, BASIC simplifies the process by taking care of the messy calculations required to project a three-dimensional object accurately on a two-dimensional screen.

FLEXIBILITY

Programmers in all languages have long been aware of the advantages of writing their own programs instead of relying on commercially available software. Of course, there are trade-offs. A commercial program is designed to do a particular application and to do it well, but it costs money. A home-brewed program takes time to write, but it is free. A commercial program cannot generally be modified to meet individual requirements. A home-brewed program can be written to your exact specifications and customized to your heart's delight.

Writing your own applications programs may sound like a formidable task, but the graphics statements available in Microsoft BASIC make it easy to write powerful graphics programs. This book will show you how.

Macintosh, the Graphics Machine

The Macintosh is one of the first computers designed with graphics in mind. To understand how it supports your graphics efforts, let's look briefly at what makes the Macintosh unique.

First, the Macintosh displays information on the screen in a novel way. Rather than the standard arrangement of rows and columns of text, the Mac screen is addressed as a matrix of tiny screen elements called *pixels*. This orientation not only allows the computer to mix graphics and text on the same screen but also lets it display text in a variety of different fonts, sizes, and styles.

This graphics orientation is supported by a very powerful set of built-in graphics routines, known as the QuickDraw graphics package. These routines, permanently stored in the Macintosh's read-only memory (ROM), create fast and efficient graphics. Microsoft BASIC allows you easy access to a healthy subset of these ROM routines via the CALL statement. You will use these routines throughout the book.

There is more to Macintosh graphics than displaying information on the screen. The printer interface is designed to accurately reproduce the Mac screen onto the Imagewriter printer. Anything you display on the screen can be printed on the printer by pressing the keys SHIFT-COMMAND-4. (The COMMAND key is the cloverleaf ⌘.)

Microsoft BASIC adds two additional ways to use your printer. You can print your entire program listing with the LLIST command. You can also use the WINDOW OUTPUT command to print graphics as large as your output device will allow. Chapter 7 gives you a complete overview of printing graphics with BASIC.

Last but not least, the Macintosh mouse is a very convenient device for graphics input. It lets you easily select objects on the screen to move them, change their orientation, give commands through dialog boxes, or choose options from a list—in short, it lets you interact with the graphics you create as you create them, as well as when you use them.

Getting Started with Microsoft BASIC

For those who are new to the Macintosh and BASIC, we include a short session on getting started. Experts should skip to the next chapter.

How do you get started with BASIC? First, make sure you have the right program. The examples in this book are created with Microsoft BASIC version 2.0. Macintosh BASIC owners can follow along with some help from the user's manual. Make a copy of your BASIC master disk either by using the Diskcopy program provided with your System Master disk or by dragging icons. The backup copy will be your working disk. Before you put the original disk away, you may want to make an extra copy, just in case.

Insert your working disk and display the directory window. Double-click on the Binary BASIC icon (the one with the π symbol). We will be using this version of BASIC for graphics because it is generally faster than Decimal BASIC.

Enter the following program in the List window:

```
RANDOMIZE TIMER
WHILE MOUSE(0)=0
x=500*RND: y=300*RND: s=5*RND
FOR red=1 TO 60*RND STEP s
  CIRCLE(x,y),red,s
NEXT red
WEND
```

You can enter the text in lowercase if you prefer. When you press RETURN at the end of each line, the computer automatically converts the key words to uppercase and displays them in boldface print. If you make a mistake, press BACKSPACE to delete the previous letter.

Click the mouse button to position the cursor for editing. You can even delete blank lines by positioning the cursor at the beginning of the next line and pressing BACKSPACE.

PRINTING THE SCREEN

When your text matches the listing, select Start from the Run menu. The program will continue to draw circles on the screen. Figure 1-9 shows a sample run.

While the patterns are in motion, let's get a printout of the output window. Make sure your printer is connected and turned on. Check the CAPS LOCK key to make sure it is up. Then press the SHIFT-COMMAND-4 keys simultaneously. (The COMMAND key is just to the left of the SPACE BAR.) This action prints a copy of the output window on paper. If you press these same keys with the CAPS KEY down, you get a printout of the entire screen. This method of copying the screen is the easiest way to get a quick printout of your program's output.

The program will run continuously until you press the mouse button. Try it now. The computer replies with **Program stopped** and re-displays the List window.

PRINTING A PROGRAM LISTING

At this point you probably want to print a listing of the program on your printer. There are two ways to do this. If the program listing fits

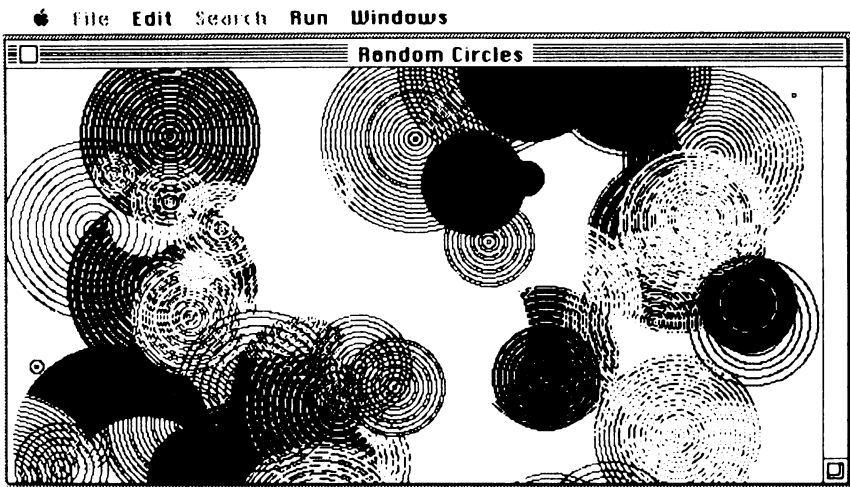


Figure 1-9.
Random concentric circles

completely in the List window, you can use SHIFT-COMMAND-4 to print the current window. For longer listings, you'll have to enter a command using the Command window. To see this, select Command from the Window menu. Type list into the Command window and press RETURN.

```
RANDOMIZE TIMER
WHILE MOUSE(0)=0
  x=500*RND: y=300*RND: s=5*RND
  FOR rad=1 TO 60*RND STEP s
    CIRCLE(x,y),rad,s
  NEXT rad
WEND
```

Notice the cramped text lines in the printout. This effect is caused by the SHIFT-COMMAND-4 command sequence, which shortens the normal line-spacing setting of the printer in order to print the output window as continuous graphics. To reset the line spacing for text, turn the printer off and then on. Now try list again.

SAVING YOUR PROGRAM

To save the current program, select Save from the File menu. The computer will ask you for the name of the file. Type Circles, and press RETURN.

OPENING ANOTHER PROGRAM

Now let's try opening a different BASIC program. Select Open from the File menu. The resulting dialog box displays a list of BASIC files on that disk. You can scroll through these files and select the one you want. You could even eject this disk and select a program from another disk, but don't do so now. For a real treat, select Music. Chapter 5's topic is music and sound, and it will show you how to use sound to enhance your graphic programs.

When you are done with that program, select Stop from the Run menu. To end the session, select Quit from the File menu. This exits BASIC and brings back the desktop. The final step is to select Eject from the File menu and turn off your Macintosh. You have just completed a successful session with BASIC.

Summary

You now have some sense of how the Macintosh can make working with graphics easy and also of the kinds of graphics you can do with

12 Macintosh Graphics and Sound

BASIC on the Macintosh. The program examples that have been presented so far only scratch the surface of what you can do. In the next chapter you will learn how to display text of all sizes and shapes on the screen. You'll also explore some of the things you can do by plotting points.

2

Printing and Plotting

In this chapter we will take a closer look at some aspects of the Mac's graphics medium, the video display—how big it is, how to print what you see on the screen onto paper, and how Microsoft BASIC displays graphics. You will also use the two most basic tools of all those you will learn in this book: printing text characters and plotting points.

The 20 programs presented in this chapter will help you learn about BASIC graphics. You should type in as many as you can (most are less than ten lines long). You'll find that actually working with the programs on the computer will give you a better understanding of the programming concepts than you would gain by simply looking at the listings in the book. The longer listings are available on a program disk to save you the trouble of typing them in.

A Look at the Video Display

The video display (or screen) can be manipulated to produce graphics images in complex ways. To control computer graphics, you

must understand the limitations of the medium. Thus it is fitting that your quest for better graphics begin with a study of the video display.

DISPLAY LAYOUT

The video display is composed of rows and columns of tiny, square areas called *pixels*. The images on the screen that you perceive as text and graphics are nothing more than carefully arranged patterns of black and white pixels. The active portion of the display is 7 1/9 inches (horizontal) by 4 3/4 inches (vertical). There are about 72 pixels per inch, both horizontally and vertically, which results in a working grid of 512×342 pixels.

Graphics created with BASIC are displayed in a portion of the screen called the *output window*. Its default size is 491 pixels wide by 254 pixels high. You can expand and compress this window with the size box to fit your needs. Windows can also be manipulated with the WINDOW statement.

Don't forget that the video display is only part of the area that is addressable with BASIC graphics statements. As you'll see later, there may be times when you want to address a much larger area (for example, when you print your graphics on a printer).

REPRODUCING THE VIDEO DISPLAY

There are two ways to transfer your graphics images to paper. The Macintosh can copy screen images to the printer with SHIFT-COMMAND-4. BASIC also lets you link the current output window to the printer with the WINDOW OUTPUT # statement. Using this technique, you can print graphics images as large as the output device will allow. See Chapter 6 for details.

On the original Imagewriter, BASIC can address an area 640 dots wide by 752 dots high (8 inches by 10 4/9 inches). To see how this area compares to the screen display, look at both parts of Figure 2-1.

The Macintosh designers made every effort to let you produce printouts that accurately reproduce what is on your video screen. The accuracy of the reproduction depends, of course, on the output device and the printing method used. For example, using the Imagewriter and the screen dump keys SHIFT-COMMAND-4, you can transfer the screen image to the printer. When you use a printer and BASIC's window print feature, however, shapes on the screen are slightly distorted as they are transferred to the printed page.

Why does this happen? The Macintosh's screen has an aspect ratio of 1 to 1, meaning that the pixels are spaced the same distance apart

horizontally and vertically. This is almost, but not quite, true of all printers' capabilities. The standard printer dot spacing on the Image-writer is 80 dots per inch (dpi) horizontally to 72 dots per inch vertically, so the aspect ratio is 10 to 9, or 1.1 to 1, instead of 1 to 1. On other printers the dot spacing may be slightly different. Thus screen images tend to be slightly compressed or expanded horizontally when they are printed, as is shown in Figure 2-2.

This slight distortion is insignificant in most cases, but if you need more precision in your printouts, you can simply adjust the image on the screen by using BASIC's GET and PUT statements. You'll see how in Chapter 6. Briefly, you use the GET statement to bring a portion of the screen into memory. Next, you use the PUT statement to bring it back to the screen in a format that is slightly stretched horizontally. Then, when the image is slightly compressed horizontally by the printer, it will print out in a 1-to-1 aspect ratio.

CHANGING THE SIZE OF YOUR WINDOW

The default output window is fine for most applications, but you may want to adjust its size. For example, you may want to extend the window to its full height of nearly 300 pixels by dragging the size box in the lower-right corner of the window. (The output window returns to the default size each time you enter BASIC.)

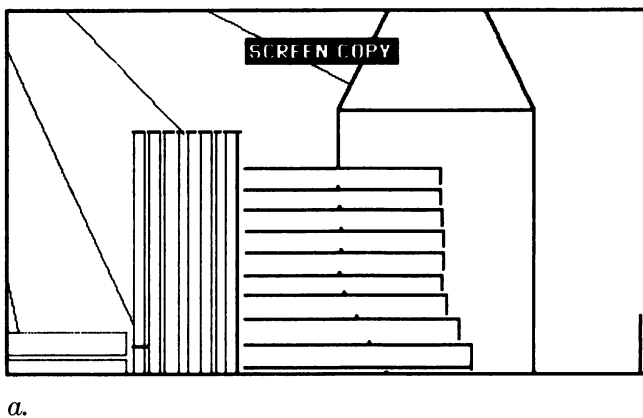
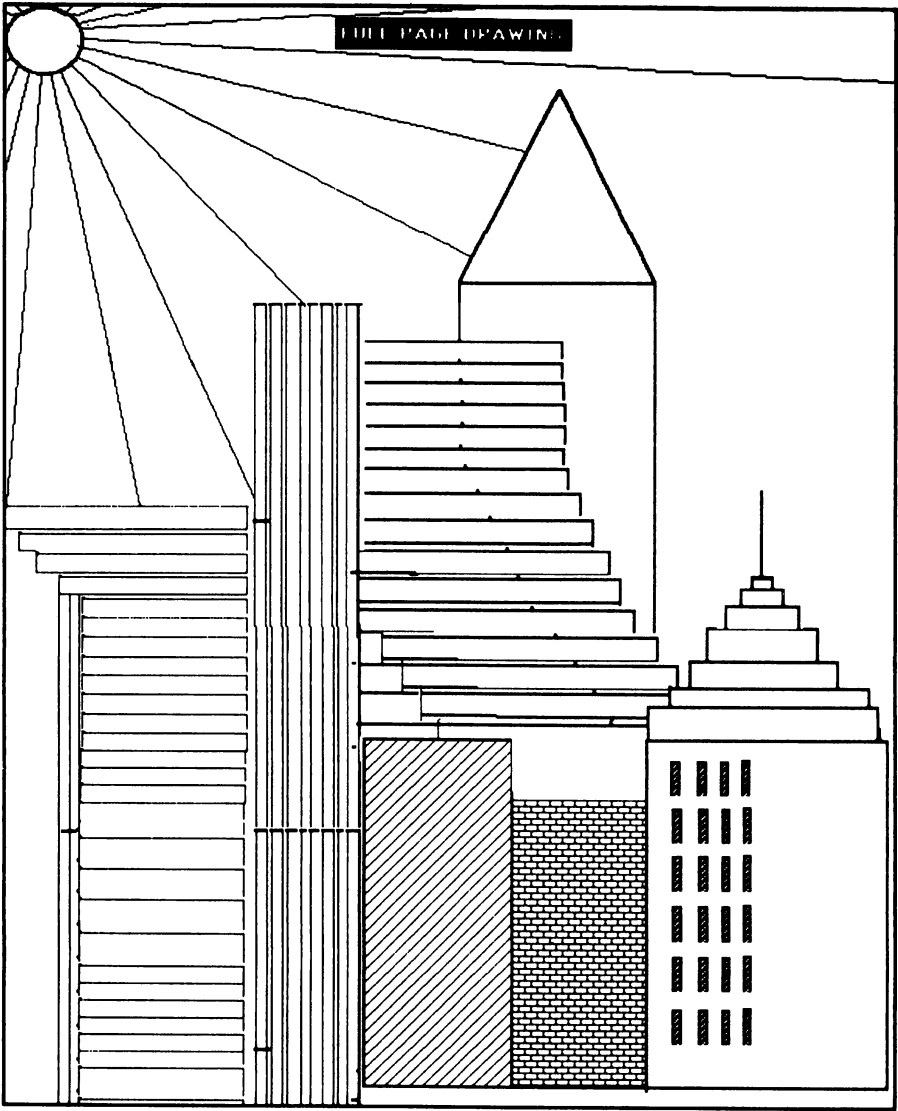


Figure 2-1.
Screen copy (a) and full-page drawing (b)



b.

Figure 2-1.
Screen copy (a) and full-page drawing (b) (*continued*)

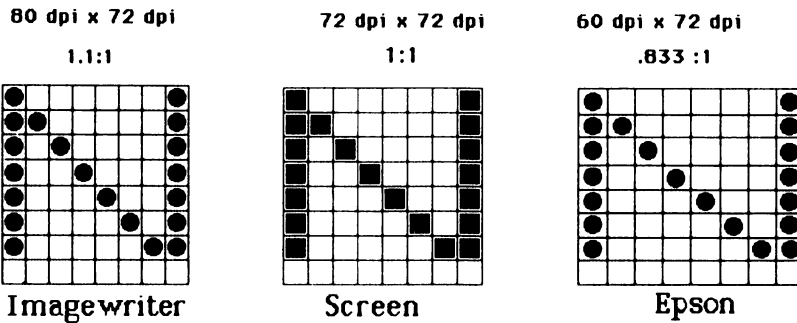


Figure 2-2.
Aspect ratio comparison

When you initiate a BASIC program run, the output window is automatically cleared to receive your new creation. You can also clear the window under program control with the `CLS` (clear screen) statement.

Plotting Points

With a clear picture of the video display, you are now better prepared to take the next step: plotting points on the screen. A pixel (or screen point) is located by its horizontal and vertical distance along two axes. Just as in high school algebra, the horizontal distance is the x coordinate of the pixel, and the vertical distance is the y coordinate. Thus each pixel position is associated with a pair of numbers, usually shown in parentheses as (x,y).

There are differences between the Macintosh coordinate system and the one you learned in high school, however. First of all, the (0,0) position (or origin) of the Mac system is *not* in the middle; it is located in the upper-left corner of the output window (see Figure 2-3). Also, the vertical values increase as points move *down* the screen. The coordinates of the visible screen area therefore go from (0,0) in the upper-left corner to (490,253) in the lower-right corner of the output window. The point (245,126) will be used as the center point for programs in this book.

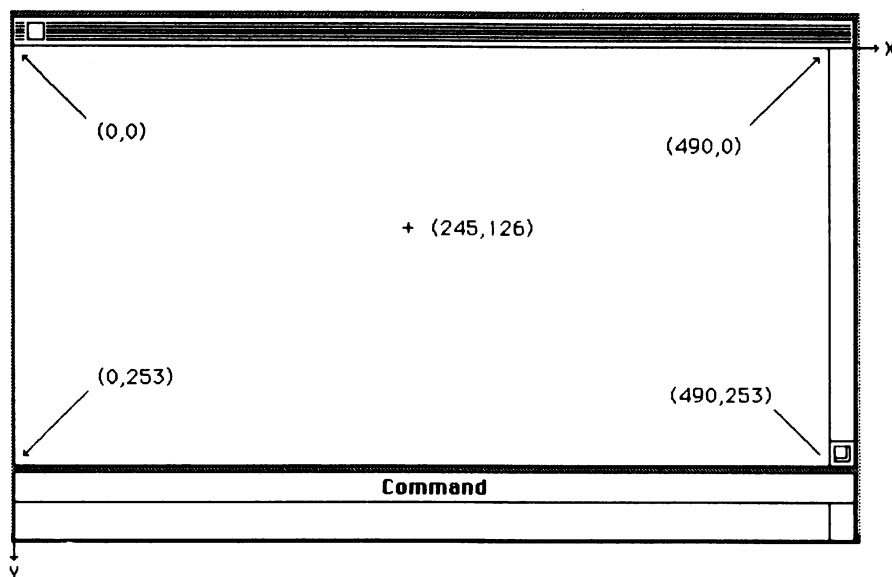


Figure 2-3.
Addressable BASIC area

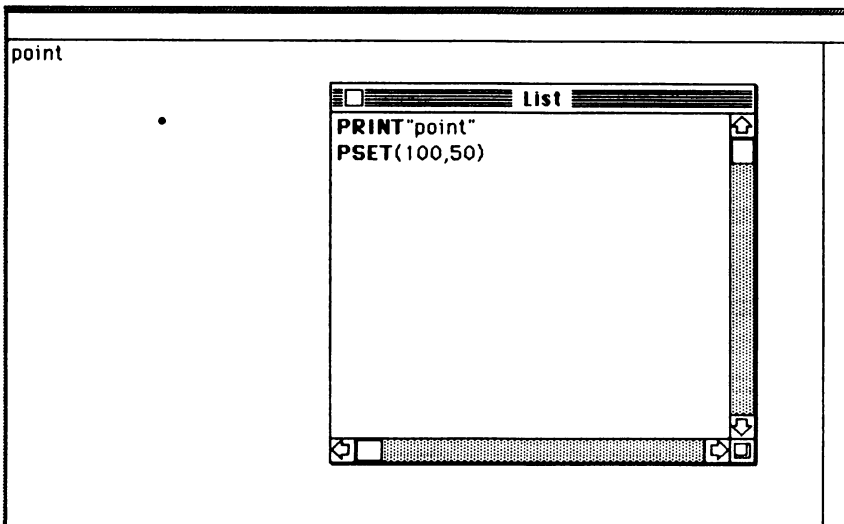
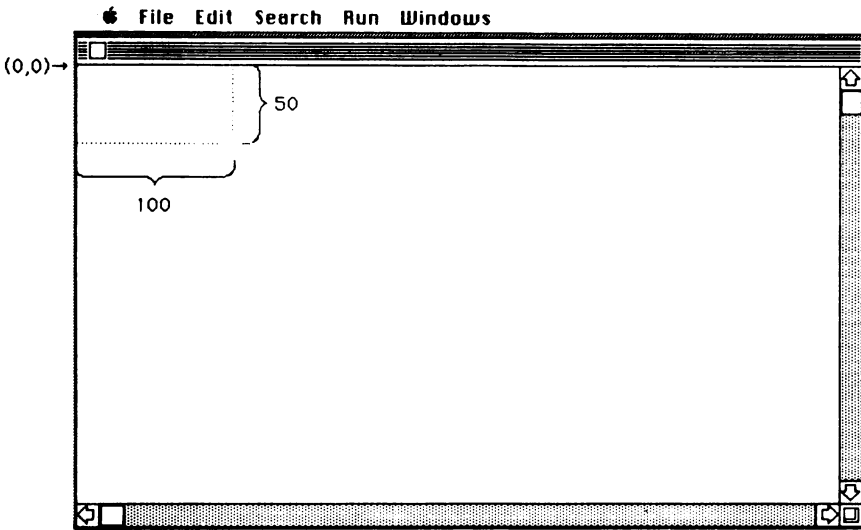
BASIC actually lets you refer to pixel positions from -32768 to 32767 in both horizontal and vertical directions. There are several advantages to having such a large drawing area, even though only a small portion of it is visible on the screen. You can let your drawings wander off the screen without getting an error message. You can also construct images and print them on output devices that accept larger formats than the video display. You will learn in Chapter 6 how to fill an entire printed page with graphics.

A SAMPLE PLOTTING PROGRAM

Let's find out how to plot points onto this grid. First, load in the binary version of BASIC [Microsoft BASIC (b)], or if you are currently using BASIC, select New from the File menu. Then type the listing shown next into the List window. (Refer to the sample session in Chapter 1 for information about entering and editing a BASIC program.)


```
PRINT "point"  
PSET(100,50)
```

To run the program, select Start from the Run menu. The PRINT statement displays the word "point" on the screen. The point-plotting statement is PSET(100,50). This statement places a black dot 100 pixels horizontally and 50 pixels vertically from the upper-left corner of the output window.



ADDING RANDOMNESS TO YOUR GRAPHICS

The List window reappears as soon as the program is finished. To make this program a little more exciting, you can add a touch of randomness by using the computer's random number function. Change the listing to match this:

RANDOMIZE TIMER

loop:

x=200+50*RND: y=100+50*RND

PSET(x,y)

GOTO loop

Again, select Start from the Run menu. The program executes in a continuous loop that will run forever unless you intervene. To exit the program, select Stop from the Run menu.

The Macintosh has a built-in random number generator that is useful for adding random action to graphics displays. It will be used frequently throughout this book. The random number generator is essentially a very long list of randomly ordered numbers between 0 and 1 (such as 0.034, 0.7791, 0.2088,...). The RND function pulls numbers from this list one at a time and returns them to the program. The RANDOMIZE statement determines where in the list the RND function starts pulling numbers. The starting position can be chosen in several ways. In this program the seed number is selected from the internal system clock (TIMER). This results in a different pattern each time the program is run. You'll find the random number feature a tremendous asset in creating animation and active graphics displays.

The RND function is used in the above program to set horizontal and vertical limits on the location of each PSET point. The horizontal coordinate (x) is fixed between 200 and 250. How is this done? Recall that RND ranges between 0 and 1. Multiplying this by 50 changes the range to between 0 and 50. Adding 200 fixes the final range of possibilities between 200 and 250. Similarly, the vertical coordinate (y) is limited to the range 100 to 150. You can use this technique any time you want to select random numbers in a specified range.

SWITCHING BETWEEN BLACK AND WHITE — OR COLOR

The PSET statement is smarter than it first appears. Not only can it plot black points on a white background, but it can also plot points in color. When the color Mac is available, you will really appreciate this

feature. For now, each color selected is displayed on the screen as black or white. The default color for PSET, as you just witnessed, is black. The numbers assigned to white and black are 30 and 33, respectively. Zero and one also give you white and black. To exercise PSET's "color" ability, let's erase each black dot in the current program and, after a short delay, replace it with a white dot.

RANDOMIZE TIMER

```
loop:
x=200+50*RND: y=100+50*RND
PSET(x,y)
FOR i=1 TO 800: NEXT i
PSET(x,y),30
GOTO loop
```

Look closely as this program runs. The first PSET paints a black dot on the screen; the second PSET paints it white. The FOR/NEXT loop adds a short time delay.

PSET has a companion statement called PRESET. It is identical to PSET except that the default is white if no color is specified. The second PSET in the preceding program could be replaced by PRESET(x,y).

CREATING RANDOM PATTERNS

In these first few programs, all pixel positions have been specified explicitly; that is, for each pixel, the program specifies exact horizontal and vertical values. This is called *absolute positioning*. The PSET and PRESET statements can also position pixels *relative* to the current pixel position. This is called *relative positioning*. Try the following:

RANDOMIZE TIMER

```
PSET(250,150)
loop:
PSET STEP (4*RND-2,2*RND-1)
GOTO loop
```

The STEP option in the second PSET statement changes the meaning of the x and y coordinates; that is, the numbers inside the parentheses are added to the current pixel coordinates. For example, the expression $4 * \text{RND} - 2$ in the second PSET statement generates random numbers between -2 and $+2$. The PSET statement converts them into the integers -2 , -1 , 0 , 1 , and 2 . The integer selected is added to the previous x coordinate to determine the x coordinate of



Figure 2-4.
Random patterns

the next point. The ultimate result is an endless variety of random patterns that resemble finely detailed pencil sketches, as illustrated in Figure 2-4.

Another way to control the stepping is shown in the following listing:

```
RANDOMIZE TIMER  
x=245: y= 126  
loop:  
h= 2*RND-1:k= 2*RND-1  
FOR i=1 TO 3 + 18*RND  
x= x+h: y= y+k  
PSET(x,y)  
NEXT i  
GOTO loop
```

In this program, the FOR/NEXT loop draws short line segments. The direction is determined by h and k.

Using Arrays to Store Points

In the current program, the numbers x and y are used to plot a point and are then recalculated in the loop's next pass. Some graphics techniques require that you store these numbers in *arrays*, which are lists of numbers or strings that use a common variable name, each one associated with a unique number. In the next program you will

use a two-dimensional array in which each cell or element of the array is referenced by its row and column number.

a(row,column)

a(1,1) →	250.4100	150.6806	← a(1,2)
a(2,1) →	250.8200	151.3612	← a(2,2)
a(3,1) →	251.2300	152.0419	← a(3,2)
a(4,1) →	251.6400	152.7225	← a(4,2)
	252.0499	153.4031	
	252.4599	154.0837	
	252.8699	154.7643	
	253.2799	155.4449	
	253.6899	156.1256	
	•	•	
	•	•	
	•	•	

In this case, the array a is used to store each point that is added to the figure. Let's take the listing out for a spin.

RANDOMIZE TIMER

n=80: head=1: tail=2

DIM a(n,2)

a(head,1)=250: a(head,2)=150

loop:

h=2*RND-1: k=2*RND-1

FOR j=1 TO 5+15*RND

PRESET(a(tail,1),a(tail,2))

a(tail,1)=a(head,1)+h: a(tail,2)=a(head,2)+k

head=head MOD(n)+1: tail=tail MOD(n)+1

PSET(a(head,1),a(head,2))

NEXT j

GOTO loop

As the program listings get longer and wider, you may wish to increase the size of the List window. A fast way to do this is to double-click on the window's title bar. It expands to fill most of the screen. Another double-click brings it back to the default size and position.

Storing the coordinate points in array a enables you to go back and erase previous points, giving the effect of a snake moving around the screen. To manage this, the variables head and tail are used to keep track of the starting and ending coordinates. These pointers are moved to the end of the array and then back to the beginning with the MOD function. Figure 2-5 shows the sequence of actions.

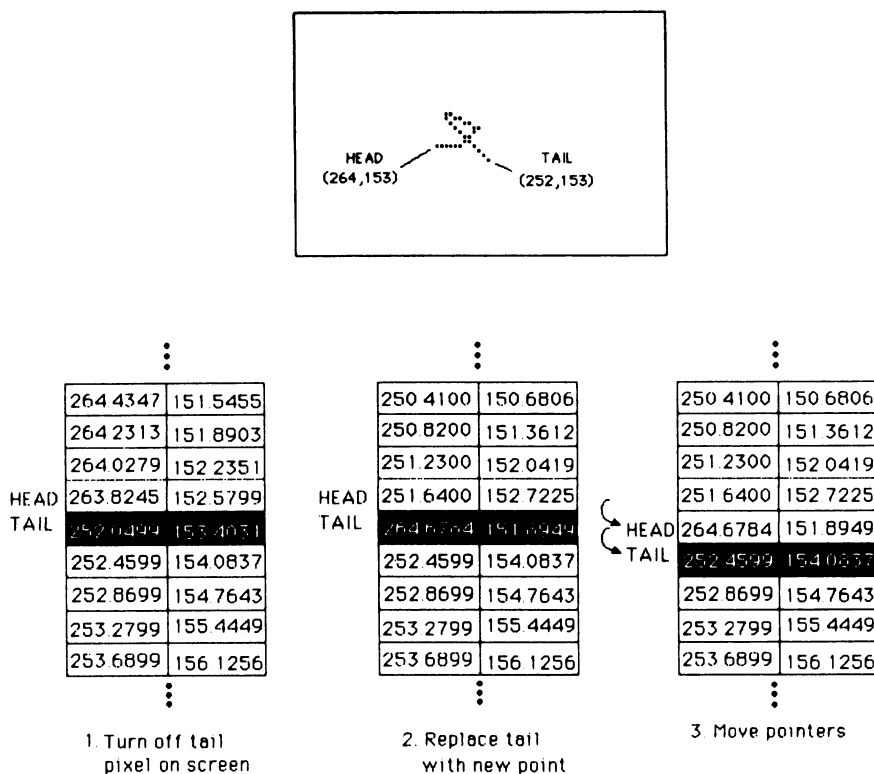


Figure 2-5.
Using array pointers

Controlling Patterns with Checkpoints

The current program runs all over the screen and crosses over itself. One way to minimize this crisscrossing is to check each point before you use PSET to make sure it is not black. Change the program to:

```
DEFINT a-z
RANDOMIZE TIMER
n=80: head=1: tail=2
DIM a(n,2)
a(head,1)=250: a(head,2)=150
FOR x=100 TO 400
```

```

PSET(x,100): PSET(x,200)
NEXT x
FOR y=100 TO 200
PSET(100,y): PSET(400,y)
NEXT y
loop:
h=2*RND-1: k=2*RND-1
FOR j=1 TO 5+15*RND
IF c>20 THEN skip
IF POINT(a(head,1)+h,a(head,2)+k)=33 THEN c=c+1: GOTO loop
skip: c=0
PRESET(a(tail,1),a(tail,2))
a(tail,1)=a(head,1)+h: a(tail,2)=a(head,2)+k
head=head MOD(n)+1: tail=tail MOD(n)+1
PSET(a(head,1),a(head,2))
NEXT j
GOTO loop

```

The POINT function returns the color value of the selected point. If it is black (that is, 33), then another point is selected. This function also prevents the snake from escaping the rectangular boundary added at the beginning of the program.

Speeding Up Your Program

Another newcomer to this program is the DEFINT statement, which changes the default variable type to integer.

Microsoft BASIC uses three types of numeric variables: integer, single-precision, and double-precision. *Integer variables* use only 2 bytes of memory and work only with integer values (-2, 0, +5, and so on); decimal numbers are converted to integers when they are stored in an integer variable. *Single-precision variables* use 4 bytes of memory and contain decimal numbers up to 7 digits long in Binary BASIC. *Double-precision variables* require 8 bytes each and are used for calculations involving decimal numbers with many significant digits. Variable types can be defined with the DEF function or can be defined individually by appending the symbol ! for single precision, # for double precision, and % for integer.

DEFINT a-z is used in the previous program to change all variables beginning with the letters a through z to integer variables. The computer can process integer variables much more quickly than single- or double-precision variables; using integer variables can help you speed up your graphics programs. However, be very careful which variables you assign as integer variables. Numbers used for decimal calculations will be truncated to integer types. If this is not

your intent, the results can be a bit bizarre. In the current program, using integer variables restricts the angles of motion to multiples of 45 degrees.

Because this snake can get boxed in, a counter (c) has been added to the program. If the program can't find a suitable direction after 20 tries, it skips the point check and goes on as if nothing were amiss.

Working with Text

The vast majority of personal and business computers are oriented toward producing lines of text on the screen. A typical system can display up to 24 lines of 80 characters. On these machines, graphics abilities have often been added as an afterthought. The Macintosh designers took an entirely different approach—the Mac screen is designed for graphics. Even text is displayed as a grid of pixels. There is no fixed number of rows and columns of text characters; it all depends on the size of the text. Of course, the system has some built-in rules about character positioning.

FONTS AND SPACING

The graphics orientation of the Mac gives you tremendous flexibility in displaying text. With BASIC, you can control four text attributes: font, size, face, and mode. The *font* is the style of a set of characters, including its upper- and lowercase letters, numbers, punctuation, and symbols. The default font for BASIC output is called Geneva. The *size* of a letter is measured in points, one point being roughly 1/72 inch. The standard size of the Geneva font is 12 points. The *face* is any of seven stylistic variations within the font—bold, underline, italic, shadow, and so on. *Mode* is the way text interacts with the current contents of the screen; for example, whether it overlays existing pixels or combines with them. Each of these attributes will be explained in detail later in this section.

To make full use of the built-in Macintosh text characters, it is important that you understand the elements that contribute to the design of a character set. Each Macintosh character is stored as a matrix of dots. The matrix must include room for upper- and lowercase characters, lowercase ascenders and descenders, spacing between lines (*leading*), spacing between characters (*letter spacing*), spacing between words (*word spacing*), and a reference point used for positioning the character on the screen. Figure 2-6 shows several elements of character design.

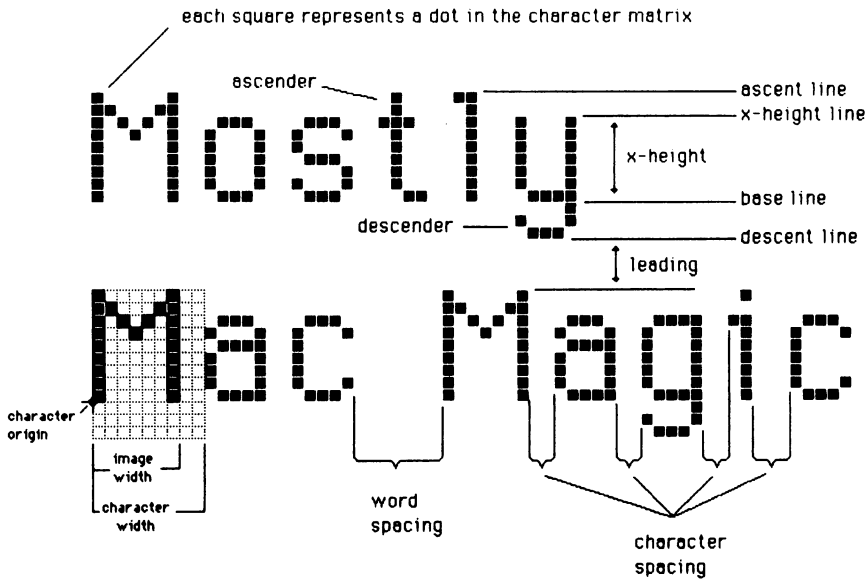


Figure 2-6.
Elements of character design

The letters shown in Figure 2-6 are 12-point Geneva characters. The *font size* is the distance between the ascent line of one row of text and the ascent line of the next row of single-spaced text. Fonts are installed on the disk in a specific point size. If you ask for a different size, the Mac scales the font up or down to match your request. Scaling fonts to odd sizes can produce unpleasant results, however. If you use odd sizes frequently, you may need to install a font size that's closer to the one you want. You can do this by using the Macintosh application Font Mover. (See "Transferring Fonts" later in this chapter for more on Font Mover.)

The character width assigned to a character depends on the shape of the image. For example, the character width of *i* is less than that of *m*. All of the original Macintosh fonts except Monaco share one characteristic: the characters' widths are spaced proportionally to the image shape. Thus, these fonts are called *proportionally spaced fonts*. Monaco, on the other hand, is a *monospaced* font, meaning that

the character width is the same for all characters, regardless of the character shape.

For text characters, the image width is generally less than the character width in order to provide spacing between adjacent characters. Border lines and other special symbols may use the entire character width so that they can join adjacent characters to form larger shapes.

The computer also puts spaces (leading) between lines of text. Leading is measured from the descent line of one row of text to the ascent line of the next. The leading for single-spaced, 12-point Geneva characters is 4 pixels.

The *character origin* is a point of reference used to position the character. It is usually placed at the left edge of the base line.

Each character of a font is defined by its dot pattern and associated spacing information. Using this information, the Mac can properly adjust for both vertical and horizontal spacing between characters. This makes text handling easier for programmers. But with so many different sizes of text fonts available, it is difficult to predict just how many rows and columns of text can be displayed on the screen for a given font size. Figure 2-7 can help you determine the number of rows and columns available for several different font sizes.

POSITIONING TEXT

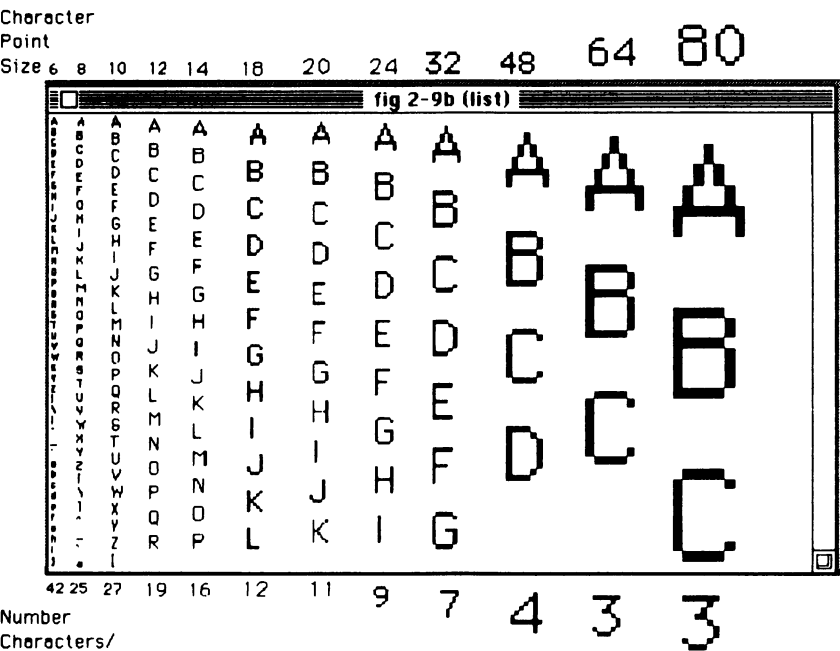
If there are no fixed positions for text on the screen, how do you control text position? There are primarily two ways to position text. One is by specifying the pixel location of the character origin point illustrated in Figure 2-6. To use this method, you must know the point size and average character width of the font and size you have selected.

A second way to position text is to locate it in terms of rows and columns. For example, you could position a character ten rows from the top and 20 columns from the left border of the current output window. Note that these rows and columns are measured in terms of the currently selected text size. For example, the screen holds 27 rows of 10-point text, but only 18 rows of 20-point text. Figure 2-8 compares the two methods of positioning text. Look at the following listing to see the difference:

```
FOR i=1 TO 12
LOCATE 1,12: PRINT CHR$(215);
NEXT i
FOR i=2 TO 12 STEP 2
```



a.



b.

Figure 2-7.
Text size examples in rows (a) and columns (b)

```

LOCATE 12,i:PRINT CHR$(215);
NEXT i
LOCATE 12,13
PRINT"LOCATE 12,12"
MOVETO 200,100
PRINT "MOVETO 200,100"
FOR i=0 TO 100 STEP 2
PSET(200,i)
NEXT i
FOR i=0 TO 200 STEP 2
PSET(i,100)
NEXT i
z: IF INKEY$="" THEN z

```

LOCATE works with rows and columns in the current text size; MOVETO uses horizontal and vertical pixel coordinates. LOCATE 12,12 positions the pen in the twelfth row and twelfth column of text. MOVETO 200,100 moves the pen 200 pixels right and 100 pixels down from the upper-left corner of the output window.

There are a couple of things you should observe in the previous program. First, the diamonds are printed using the CHR\$ function. This handy function can be used to print any of the symbols shown in Appendix A of the *Microsoft BASIC Manual*. Second, the last line of the program is used to pause the output display until any key on the keyboard is pressed. The program list is then displayed. The INKEY\$ function is the key element here. This function constantly

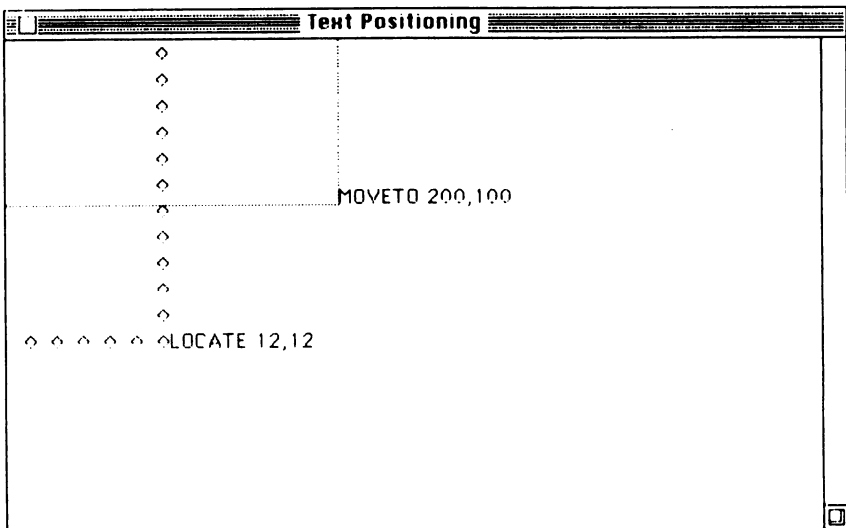


Figure 2-8.
Two methods of positioning text

strokes the keyboard for input. Each time a key is pressed, the INKEY\$ function remembers the key's value. But there is only room in INKEY\$'s memory for one keystroke. So if INKEY\$ is empty (that is, INKEY\$=""), then no keys have been pressed since the last time it was used. This statement stays in a loop until any key is pressed, which gives you an easy way to protect a graphics display from being disrupted by overlapping windows.

SELECTING TEXT

Now that you know how to position text, how do you select the type of text you want? Four main text attributes are available through calls to the Macintosh ROM: font type, size, face, and mode. The following discussion introduces these text attributes one at a time. Use Figure 2-9 as a reference throughout the discussion.

Text Font

Your BASIC system disk comes with three fonts: Chicago, Geneva, and Monaco. Actually, there is more to it than that. Each font stored on the disk is associated with a specific point size. The system disk thus contains Chicago-12, Geneva-9, Geneva-12, and Monaco-9.

What about all the other wonderful fonts that are supplied with your Macintosh? They are stored in the Fonts and System files located on the System Master disk. (There are also commercially available fonts. See your Macintosh dealer for more information.)

Transferring Fonts

To transfer fonts from any Mac disk to the BASIC system disk, save any programs to disk; then reset the computer. Insert the BASIC system disk in the internal drive and the System Master disk in the external drive (single-drive users will have to do some disk swapping). Double-click on the Fonts/DA Mover icon on the System Master disk. (*Note:* if you are still using the old Font Mover program, your Macintosh dealer can provide you with the improved Font/DA Mover program). This activates the Font/DA Mover program that allows you to transfer fonts between the System file on your BASIC disk and a Fonts file on any of your other disks. Click on the Close button corresponding to your System Master disk. Click on the Open button to display the standard Mac file dialog window. Eject your System Master disk (you may have to click the Drive button first) and insert a disk containing the fonts you want to transfer. Double-click on the file you wish to use.

To make the transfer, select a font from the window and click the Copy button to transfer the font to the System file of your BASIC

	FONT	FACE	SIZE	MODE
0	Chicago	Plain		Copy
1	Geneva	Bold		OP
2	New York	<i>Italic</i>		XOR
3	Geneva			Bit
4	Monaco	<u>Underline</u>		
5	Venice			
6	London		8	
7	Rhodes		7	
8	San Francisco	Outlined	8	
9	Toronto		9	
10	Seattle		10	
11	Cairo(☺↑)		11	
16		Shadow	16	
24			24	
32		Condensed	32	
64		Extended	64	
127			127	

Figure 2-9.
Text attributes

disk. If you need help, select the Help button or check *Macintosh*, your owner's manual, for assistance.

Once you have all the fonts you want on your BASIC disk, how do you select a text font from BASIC? Use the TEXTFONT ROM call. Reopen BASIC (binary version) and enter the following program:

```
FOR i=0 TO 4
TEXTFONT i
PRINT "text font";i
NEXT i
```

TEXTFONT 1**PRINT "Back to the standard font"****z: IF INKEY\$="" THEN z**

(Press any key to end this program.)

Check your output with the first column of Figure 2-9. Font 0 is Chicago. This is the font used by the Macintosh (look at the titles in the menu bar). Font 1 is Geneva. This is the font chosen as the default for BASIC. Font 2 should be New York, but if that font is not resident on the system disk, BASIC uses the default font (Geneva) instead. Font 3 is Geneva. Font 4 is Monaco, the only monospaced font in the Mac.

Caution: Be sure to restore the default text attributes before leaving any program. BASIC does not revert to the defaults at the start of each program you run, so any text attributes active at the end of one program run will be retained at the start of the next. Once you start manipulating text attributes, you assume responsibility for keeping track of the active ones.

Font numbers 0 through 127 are reserved for fonts provided by Apple. The numbers 128 through 383 are reserved for selected software vendor fonts, and 384 through 511 are for use by the rest of us. If you plan to create your own fonts with a font utility, be sure you use numbers in the appropriate range.

Text Size

Take another look at text font 4 (Monaco) in the last program run. Something is definitely wrong. The problem is that the Monaco font is stored on disk as size 9. The program is asking the Mac to print in size 12, the default text size. When you select a text size that differs from the available sizes in the System file, the Mac selects the closest size of that font and then stretches or compresses it to match the requested size. In this process, the characters may become distorted. Look back at Figure 2-7 to see what happens to Geneva characters when they are stretched to various sizes.

Let's try out a few different text sizes. Notice that the computer handles line spacing in terms of the current text size. Try the following:

TEXTFONT 4**TEXTSIZE 9****FOR i=1 TO 4****PRINT "Monaco - 9"****NEXT i**

```

TEXTSIZE 18
FOR i=1 TO 4
PRINT"Monaco - 18"
NEXT i
TEXTFONT 1
TEXTSIZE 12
z: IF INKEY$="" THEN z

```

Notice that the first Monaco-18 prints right over the fourth Monaco-9. The linefeeds are all done in the current text size, so the linefeed after the last Monaco-9 is too short to make room for the first Monaco-18.

The limits on text size are 1 to 127 points. Sizes less than 6 points are too small to be useful and may result in a system error.

Text Face

Figure 2-9 shows the seven different ways in which text can be manipulated to have different faces. What it doesn't show is that these text variations can be used in combination as well. That's why the Face column is shaded. To select a combination of two or more faces, simply add up the values associated with each face. Enter the following:

```

TEXTSIZE 18
TEXTFACE 65
MOVETO 70,60
PRINT "TEXT"
TEXTSIZE 12
TEXTFACE 0
z: IF INKEY$="" GOTO z

```

The TEXTFACE(65) call activates the combination of bold (1) and extended (64) as shown in Figure 2-10. Most of the text faces in the figure are self-explanatory, but expanded and compressed simply stretch or compress space between letters. Note that outline (16) and shadow (8) don't work in the default text mode; we'll see why below.

Text Mode

The last text attribute is called text mode. It governs the way text will interact with the pixel patterns already on the video display. A mode is selected with the TEXTMODE call, followed by a number from 0 to 3. The default mode (called copy) causes text to overlay anything that is currently on the screen. The other three modes are labeled OR, XOR, and BIC (Black Is Changed). Figure 2-11 shows how the word *text* interacts with three different backgrounds (black, white, and gray) for each of the modes.

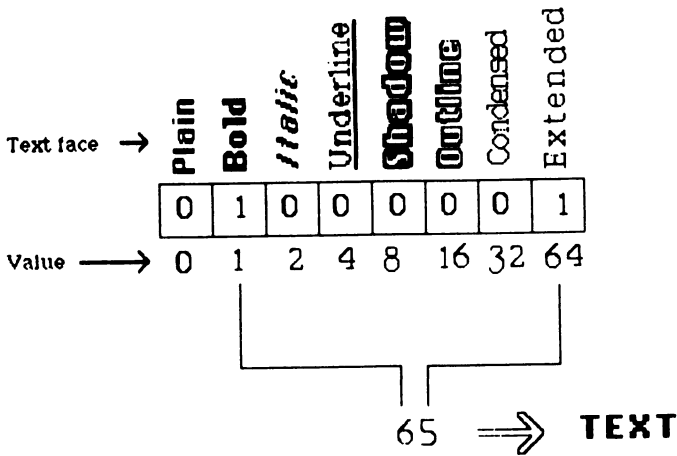


Figure 2-10.
Text-face combinations

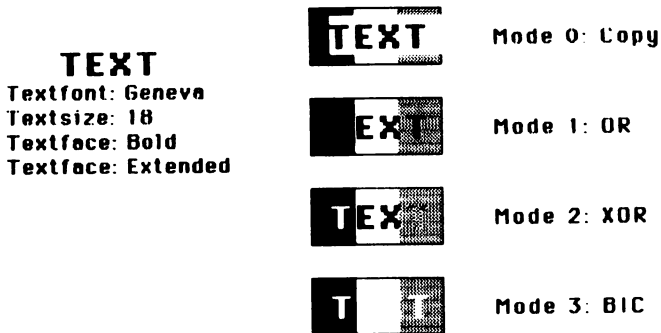


Figure 2-11.
Text modes

Mode 0 is the copy mode, which simply overlays the backgrounds. Each letter is surrounded by a white rectangle that obliterates anything currently on the screen. This creates a useful frame around text that is printed on a solid black background. Note that shadow and outlined attributes do not work in Mode 0.

Mode 1 is called OR. In this mode the text characters are transparent; that is, if either the overlaying text character or the screen has a black dot at a given pixel location, the result is black. The only white areas are where both the text and the screen are white.

Mode 2 is labeled XOR, which stands for exclusive OR. In this mode you get white if both text and screen pixels are white and also if they both are black. If either the text pixels or the screen pixels are black (not both), you get black. For an example, look at what happened to the first T in "TEXT." Both the T and the background pixels are black, so the result is white. A good way to think of mode 2 is to remember that all the black dots in the text will invert the existing dots on the screen.

Mode 3 is called BIC, for Black Is Changed. In this mode, black dots in the text turn white and replace the existing pixels. Note that shadow and outline faces do not work in mode 3.

QuickDraw Routines and ROM Calls

The LOCATE, MOVETO, and TEXT statements covered earlier in this chapter are examples of the many QuickDraw routines stored in the read-only memory (ROM) of the Macintosh. These routines can generally be accessed with the CALL statement. For example, LOCATE can be executed as CALL LOCATE(r,c). Fortunately, Microsoft BASIC lets you drop the CALL and the parentheses for all ROM calls except LINE, since LINE is also a BASIC key word. The programs in this book do not use the word CALL or the parentheses when using these routines. Thus, in appearance, ROM calls are virtually indistinguishable from their BASIC counterparts; there are differences, however, as you will discover throughout the book.

Applications

Most of the chapters from this point on will end with some practical applications of the graphics concepts you have learned in that chapter. Reviewing the program listings will give you a chance to see these concepts used in combination to solve specific problems. You can adapt the methods presented here to your own applications.

Some of the programs introduced in this chapter will be revised in later chapters to incorporate new graphics tools as they are introduced, so keep your eyes open for improvements.

If you're an avid programmer, you'll probably skip the beginning of

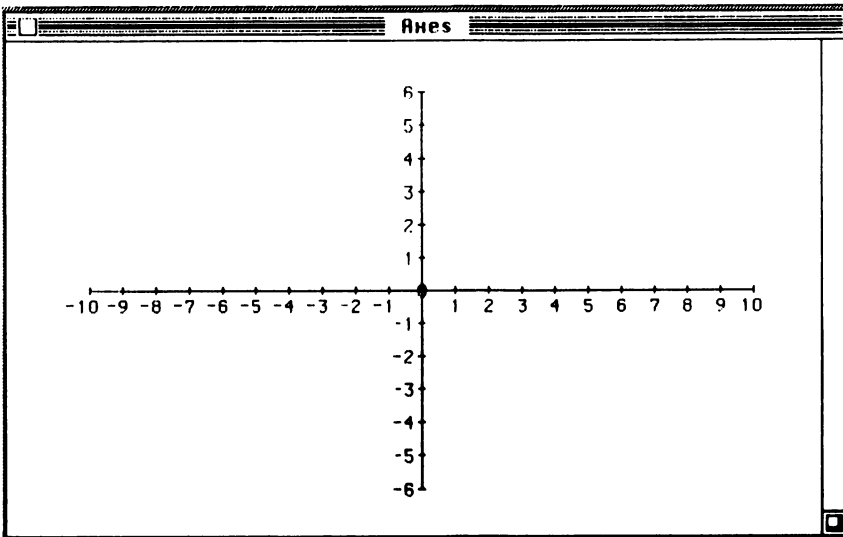


Figure 2-12.
Cartesian coordinate system

each chapter and start right in with the applications programs to find out what secrets they hold. Comments will be included to help you decipher some of the more difficult programs and to emphasize many of the graphics techniques used.

CARTESIAN COORDINATE SYSTEM

One of the most important applications for PSET is plotting functions. This program draws two coordinate system axes. Each unit on either axis is 20 pixels (see Figure 2-12).

```

TEXTSIZE 10
TEXTFONT 10
TEXTMODE 1
FOR x=50 TO 450: PSET(x,130): NEXT x
FOR x=50 TO 450 STEP 20
IF x=250 THEN endloop
FOR y=128 TO 132: PSET(x,y): NEXT y
left=10: IF x=50 OR x=450 THEN left=15
MOVETO x-left,144: PRINT (x-50)/20-10;
endloop: NEXT x
FOR y=10 TO 250 STEP 20
IF y=130 THEN MOVETO 240,135: GOTO skip
MOVETO 232,y+5
skip: PRINT 5-(y-30)/20;

```

```

FOR x=248 TO 252: PSET(x,y): NEXT x
NEXT y
FOR y=10 TO 250: PSET(250,y): NEXT y
TEXTSIZE 12
TEXTMODE 0
TEXTFONT 1
Z: IF INKEY$="" THEN Z

```

Finding enough room for the numbers was a problem here so the 10-point Seattle font was installed instead of scaling down a 12-point font. (See the discussion earlier in this chapter on transferring fonts.)

Another problem in this program is that text characters in the default mode 0 have a tendency to wipe out everything in their path. You can get around this to some extent by printing all the text first and then plotting the graphics. This is not always practical. For this program, text mode 1 is used instead, so that the text will mingle with the graphics already on the screen. Try it with mode 0 to see what happens.

FUNCTION PLOTS

With the axes prepared in the previous program, all you have to do now is plot some friendly functions. This involves adding only a few program lines:

```

TEXTSIZE 10
TEXTFONT 10
TEXTMODE 1
FOR x=50 TO 450: PSET(x,130): NEXT x
FOR x=50 TO 450 STEP 20
IF x=250 THEN endloop1
FOR y=128 TO 132: PSET(x,y): NEXT y
left=10: IF x=50 OR x=450 THEN left=15
MOVETO x-left,144: PRINT (x-50)/20-10;
endloop1: NEXT x
FOR y=10 TO 250 STEP 20
IF y=130 THEN MOVETO 240,135: GOTO skip
MOVETO 232,y+5
skip: PRINT 5-(y-30)/20;
FOR x=248 TO 252: PSET(x,y): NEXT x
NEXT y
FOR y=10 TO 250: PSET(250,y): NEXT y
FOR x=-10 TO 10 STEP .01
f:y=2*(x*x-9*x+8)/(2*x*x+x-4)
IF y>6 OR y<-6 THEN endloop2
x1=20*x+250
y1=-20*y+130
PSET(x1,y1)

```

```

endloop2: NEXT x
TEXTSIZE 12
TEXTMODE 0
TEXTFONT 1
z: IF INKEY$="" THEN z

```

Figure 2-13 shows the plotted function.

The only problem in plotting these functions is the quality of the steep vertical lines. The FOR/NEXT loop that plots the function (line f) steps along the x axis in increments of 0.01. For each x value, the y value is calculated; then both x and y are translated to screen coordinates x1 and y1 for plotting. The step increment 0.01 results in somewhat sketchy vertical lines around $x=1$. You can improve the continuity of the lines by using a smaller step increment, but the smaller the increment, the slower the program will run.

Those who are mathematically inclined can try another approach with certain functions. If a function is "one-to-one" (that is, has an inverse), you can plot the function in two passes. Make one pass along the x axis; then calculate the inverse and make a pass along the y axis. (Another alternative is to use the LINETO command introduced in the next chapter to make continuous lines.)

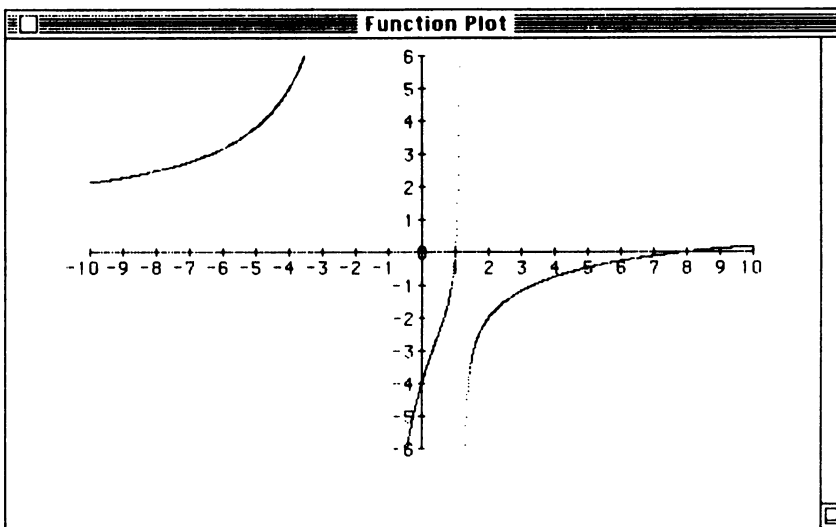


Figure 2-13.
Function plot

The beauty of this program is that once it is entered, you can plot virtually any function just by changing line f:. Try some of the following:

```
y=3*SIN(3*x) + 2*COS(2*x)
y=abs(1/4*x)
y=-x + 2
y=1/2*x^3 - 4*x
```

POLAR FUNCTIONS

Rectangular coordinates are not the only possibilities for the mathematical purist. For drawing circles and spirals, polar coordinates are much easier to work with. In a polar system, points are plotted by their counterclockwise angle from the positive x axis and their straight-line distance from the intersection (called the origin) of the two axes. Figure 2-14 shows what elegant figures can be drawn using polar coordinates.

```
FOR x=50 TO 450: PSET(x,150): NEXT x
FOR y=30 TO 270: PSET(250,y): NEXT y
FOR angle=0 TO 20 STEP .01
f: r=30*angle*COS(angle)*SIN(angle)
PSET(250+r*COS(angle), 150+r*SIN(angle))
NEXT angle
z: IF INKEY$="" THEN z
```

The length of the program listing should convince you of its simplicity. The FOR statement selects angles from 0 to 20 radians in steps of 0.01. Line f is the function that calculates the corresponding radius. The PSET line translates the radius and angle into x and y screen coordinates and plots the point.

Once again, you can replace line f: with your own function for endless variety. Here are some samples to experiment with:

```
r=60
r=3*angle
r=100*COS(3*angle/2)
r=60*SIN(2*angle)+40*COS(3*angle)
r=10*angle*COS(3*SIN(2*angle))
r=10*angle*COS(SIN(6*angle))
```

You can create one interesting variation by showing only a short segment of the figure as it is being drawn. Combine the technique given in the listing found in the "Using Arrays to Store Points" sec-

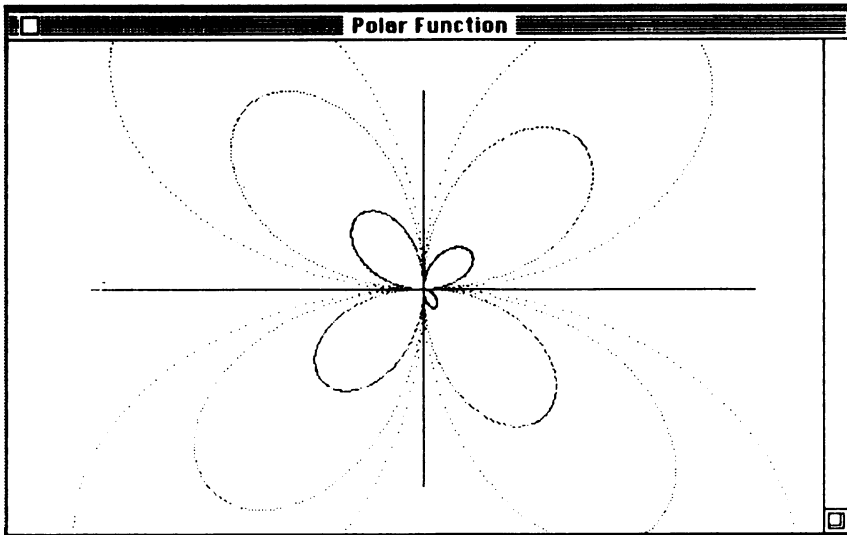


Figure 2-14.
Polar function

tion of the chapter with the current program to show a smooth curve tracing a pattern on the screen.

```
n=40: head=1: tail=2
DIM a(n,2)
a(head,1)=250: a(head,2)=130
FOR angle=0 TO 20 STEP .01
PRESET(a(tail,1),a(tail,2))
f: r=30*angle*COS(angle)*SIN(angle)
x=250+r*COS(angle): y=130+r*SIN(angle)
a(tail,1)=x: a(tail,2)=y
head=head MOD(n)+1: tail=tail MOD(n)+1
PSET (x,y)
IF INKEY$<>" THEN z
NEXT angle
z: END
```

Again, you can replace line f with any function that you choose.

Creating a Logo

One particularly challenging application of graphics is the development of a product logo or letterhead for business correspondence. In many cases it is easier to create a logo with a drawing program like

MacPaint; in others, more control of the computer is required. This book explores the possibilities of using BASIC to design logos. In this chapter our techniques are limited to plotting points and using text characters, but watch for this application in later chapters.

The first example prints oversized letters and then randomly prints white dots in a rectangle that encloses the three letters (Figure 2-15).

```

TEXTMODE 2
TEXTSIZE 40
TEXTFONT 10
TEXTFACE 16
LOCATE 1,1
PRINT "XYZ"
FOR n=1 TO 3000
x=110*RND: y=10+60*RND
PRESET(x,y)
NEXT n
TEXTMODE 0
TEXTSIZE 12
TEXTFONT 1
TEXTFACE 0
z: IF INKEY$="" THEN z

```

The second example duplicates each letter with a slight vertical and horizontal offset. Mode 2 is used to get an alternating black-and-white effect (Figure 2-16).

```

TEXTFONT 0
TEXTMODE 2
TEXTFACE 0
TEXTSIZE 100
FOR n=1 TO 8
MOVETO 2*n,120-3*n
PRINT "Zoo";
NEXT n
TEXTFONT 1
TEXTMODE 0
TEXTSIZE 12
TEXTFACE 0
z: IF INKEY$="" THEN z

```

You might also try putting the text size call in the loop to vary the text size for each pass.

EXPLODING UNIVERSE

The next program is an exercise in positioning pixels on the screen (Figure 2-17).



Figure 2-15.
Logo 1

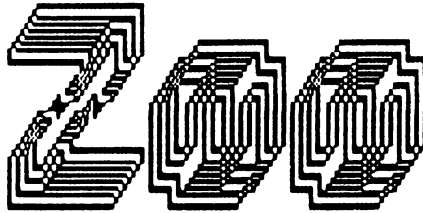


Figure 2-16.
Logo 2

```

DEFINT a-z
n=100
FOR r=150-n TO 150+n
FOR c=250-n TO 250+n
d#=((r-150)^2+(c-250)^2)/n^2
IF d#>RND(1) THEN PSET(c,r)
NEXT c
NEXT r
z: IF INKEY$="" THEN z
    
```

The program illustrates selective use of variable types. Notice that all of the variables a through z are defined as integers to increase program speed. In calculating the distance, a double-precision variable (d#) is used.

This program scans a square area and calculates the distance from each point to the center. The distance is divided by the square of *n*

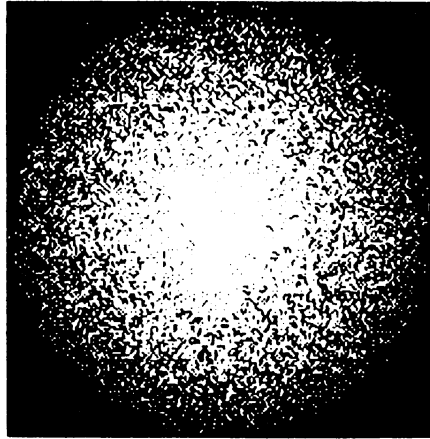


Figure 2-17.
Exploding Universe

and is then compared to a random number between zero and one. If the modified distance is greater than the random number, the point is plotted. The net effect is to give points farther from the center a greater probability of being plotted and thus produce a “big-bang” effect.

The program takes several minutes to run, which brings up the question of how to speed up a program. Program speed is discussed in detail in Chapter 10, but as a general rule, define all variables as integers whenever possible to speed program execution. Just make sure you declare variables that use decimal calculation as single-precision variables (such as `x!`) or double-precision variables (such as `d#`).

LIFE

John Conway’s game of Life attempts to simulate the growth of a colony of living cells. The births and deaths of cells are governed by these simple rules:

Survivals: Each live cell with 2 or 3 live neighbors survives.

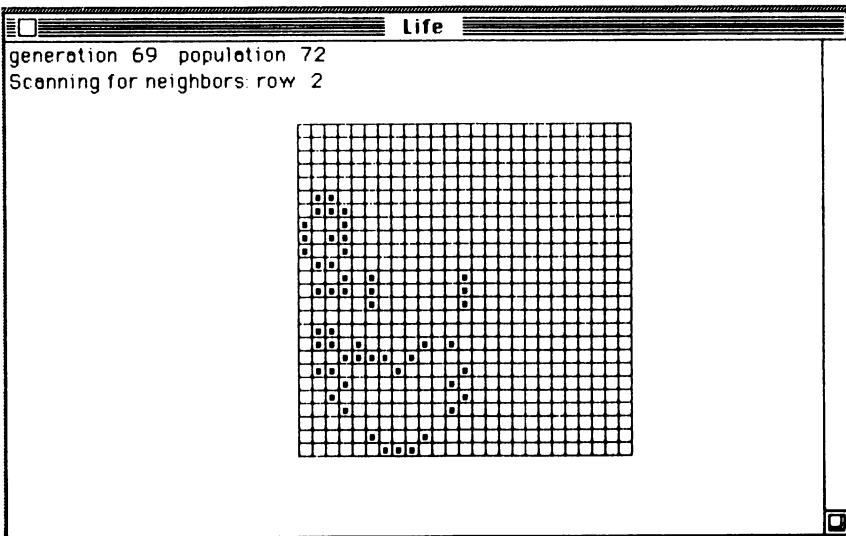


Figure 2-18.
The game of Life

Deaths: A live cell with more than 3 neighbors dies from overpopulation.

Births: An empty cell with *exactly* 3 neighbors comes alive in the next generation.

Note: Births and deaths occur simultaneously.

There is no real object to the game other than to try different patterns of cells and to observe how the population changes after each generation. Figure 2-18 shows the game board after several generations.

```

DEFINT a-z: DIM a(25,25): g=1: p=0
PRINT "initializing game board: generation: ";g;
FOR r=1 TO 25: READ s$
FOR c=1 TO 25
  a(r,c)=VAL(MID$(s$,c,1))
IF a(r,c)=1 THEN PSET(171+8*c,46+8*r): p=p+1
NEXT c
NEXT r
PRINT "population";p
FOR x=175 TO 375 STEP 8
  FOR y=50 TO 250: PSET(x,y): NEXT y
NEXT x

```

```

FOR y=50 TO 250 STEP 8
FOR x=175 TO 375: PSET(x,y): NEXT x
NEXT y
LOCATE 1,1: PRINT SPACES$(80)
LOCATE 1,1: PRINT "generation";g;" population";p
checkstatus:
p=0
FOR r=1 TO 25: LOCATE 2,1
PRINT"Scanning for neighbors: row";r
FOR c=1 TO 25: nb=0: IF a(r,c)=1 THEN p=p+1
FOR h=-1 TO +1: IF c+h<1 OR c+h>25 THEN A:
FOR v=-1 TO +1: IF r+v<1 OR r+v>25 OR h=0 AND v=0 THEN B:
IF a(r+v,c+h)=1 OR a(r+v,c+h)=2 OR a(r+v,c+h)=4 THEN nb=nb+1
B: NEXT v
A: NEXT h
IF a(r,c)=1 THEN IF nb<2 OR nb>3 THEN a(r,c)=4 ELSE a(r,c)=2
IF a(r,c)=0 THEN IF nb=3 THEN a(r,c)=3
NEXT c
NEXT r
updateboard:
g=g+1: LOCATE 1,1: PRINT "generation";g;" population";p
FOR r=1 TO 25
FOR c=1 TO 25
IF a(r,c)<3 THEN C
FOR y=45+8*r TO 48+8*r
FOR x=170+8*c TO 172+8*c
IF a(r,c)=4 THEN PRESET(x,y) ELSE PSET(x,y)
NEXT x
NEXT y
C: NEXT c
NEXT r
redoarray:
p=0
FOR r=1 TO 25: FOR c=1 TO 25
IF a(r,c)=2 OR a(r,c)=3 THEN a(r,c)=1
IF a(r,c)=4 THEN a(r,c)=0
IF a(r,c)=1 THEN p=p+1
NEXT c,r
GOTO checkstatus:
DATA "10010000000000000000000000000000"
DATA "00001000000000000000000000000000"
DATA "10001000000000000000000000000000"
DATA "01111000000000000000000000000000"
DATA "00000000000000000000000000000000"
DATA "00000000000010000000000000000000"
DATA "00000000000100000000000000000000"
DATA "00000000010000000000000000000000"
DATA "00000000100000000000000000000000"
DATA "00011111000000000000000000000000"

```

```

DATA "000111100000000000000000"
DATA "000101100000000000000000"
DATA "000000000000000000000000"
DATA "000000000000000000000000"
DATA "000000001001000000000000"
DATA "000000001111000000000000"
DATA "000000010000100000000000"
DATA "000000010110100000000000"
DATA "000000010000100000000000"
DATA "000000001111000000000000"
DATA "000000000000000000000000"
DATA "000000000010000000000000"
DATA "000000000000000000000000"
DATA "000000000000000000000000"

```

The program is driven by the strings in the DATA statements at the end of the listing. A live cell is represented by 1; 0 represents no cell. These strings are loaded character by character in the array a(r,c). After the initial grid is displayed, the checkstatus section scans the board row by row. The number of neighbors is calculated for each cell, and the cells are relabeled with 2 (survive), 3 (birth), or 4 (death). The updateboard section draws the next generation. The Redoarray section changes all the 2s, 3s, and 4s back to 1's and 0's. Then the program loops back to the checkstatus section to start the next generation.

STARSHIP VIEWSCREEN

The next program simulates stars whizzing by as you gaze out the screen of your command console. Black stars against a white background would not seem natural, so the QuickDraw call BACKPAT is used to select a black background and white dots.

```

DEFINT a-z: DEFMSG d: RANDOMIZE TIMER
BACKPAT 492: CLS
a: FOR i=1 TO 5: PSET(x(i),y(i))
IF f(i)>0 THEN PSET(x(i)+1,y(i)+1)
IF f(i)=2 THEN PSET(x(i)+1,y(i)): PSET(x(i),y(i)+1)
IF x(i)+h(i)>0 AND x(i)+h(i)<490
AND y(i)+k(i)>0 AND y(i)+k(i)<253 THEN cf(i)=0
b: x(i)=203+84*RND: h(i)=x(i)-245: IF h(i)=0 THEN b
y(i)=108+36*RND: k(i)=y(i)-126: GOTO d
c: x(i)=x(i)+h(i): y(i)=y(i)+k(i):
d=:(i-245)^2+(y(i)-126)^2
IF d>10000 THEN f(i)=2: GOTO d
IF d>2000 THEN f(i)=1

```

```

d: PRESET(x(i),y(i))
IF f(i)>0 THEN PRESET(x(i)+1,y(i)+1)
IF f(i)=2 THEN PRESET(x(i)+1,y(i)): PRESET(x(i),y(i)+1)
NEXT i
z: IF INKEY$="" THEN g ELSE BACKPAT 380: CLS

```

Be careful using BACKPAT at this point. The BACKPAT routine is designed to be used with patterns that have been stored in an array earlier in the program. You'll learn how to define your own patterns in the next chapter. For now, you can select areas of memory that give you the patterns you want. By poking around at random, you'll find that memory location 492 gives a black background and 380 gives a white background. These areas of memory should be consistent on all machines. If you experiment with different numbers, be sure to use only even-numbered locations; a system error may result if you use odd numbers. A CLS (clear screen) statement activates the selected background pattern.

The Starship viewscreen program places white dots representing stars near the middle of the screen and then moves them out to the edge. The program keeps five stars active at all times. When a star gets near the window border, it disappears, and a new star appears in the center of the screen.

The horizontal and vertical coordinates of star *i* are stored in *x(i)* and *y(i)*, where *i* ranges from 1 to 5. The angle and speed at which the star moves toward the edge of the window are determined by *h(i)* and *k(i)*.

To make the stars increase in size as they approach the edge of the window, a flag variable *f(i)* was set. For dots close to the center, *f(i)* is 0, and only one dot is displayed. For dots past a certain radius, *f(i)* is set to 1, and two dots are displayed. For dots close to the edge of the screen, *f(i)* is set to 2, and four dots are displayed. The process of increasing the size will be simplified with the tools developed in the next chapter.

To stop the program, press any key. The background pattern reverts to white in program line *z*.

Summary

In Chapter 2 we have developed some of the basic tools necessary to work with graphics on the Macintosh screen.

First, you have learned about the video display and how to reproduce it accurately on a printer. You have also learned how to plot

points. Other useful techniques you have gained include the random number function, absolute versus relative positioning, erasing points, and even rudimentary animation.

This chapter has also presented basic character design, how you can position text on the screen, and how you can use the four different text attributes available on the Macintosh: font, size, face, and mode. The chapter ended with several programs that show you different ways to use point plotting and text characters for a variety of applications.

In Chapter 3, you will learn how to draw such basic shapes as lines, rectangles, circles, and polygons. You will also learn how to fill these shapes with patterns.

The end of each chapter will include a list of the statements, functions, ROM calls, and other BASIC key words used for the first time in the programs in that chapter. Since you are already familiar with BASIC, only those terms from the list that are unique to the Mac or those that play a significant role in BASIC graphics are explained in the chapter. For this first programming chapter, the list is quite long; in subsequent chapters, it will provide a convenient summary of the new key words introduced.

BASIC Statements, Connectives, and Functions

AND	FOR	PTAB	TO
CHR\$	IF	RANDOMIZE	VAL
CLS	INKEY\$	READ	
COS	MOD	RND	
DATA	NEXT	SIN	
DEFINT	OR	SPACE\$	
DEFSNG	POINT	STEP	
DIM	PRESET	THEN	
ELSE	PRINT	TIMER	
END	PSET		

ROM Calls

BACKPAT
LOCATE
MOVETO
TEXTFACE
TEXTFONT
TEXTMODE
TEXTSIZE

3

Drawing Basic Shapes And Patterns

In this chapter you will learn to draw lines. You will also learn how to create simple geometric shapes and fill them with patterns. These techniques are the building blocks that you will use over and over to develop more complex designs and patterns.

Drawing Simple Shapes

An elementary skill of the traditional pencil artist is drawing basic shapes freehand — circles, straight lines, arcs, and so forth. You do not need to concern yourself with developing such freehand skills, because your computer already has them mastered. Your role is to tell the computer what figures to draw and where to locate them on the screen; the Mac takes care of all the details.

LINES

To draw a straight line, all you have to do is tell the Mac the two endpoints, and it does the rest. Each point is located by its horizontal and vertical distance from the upper-left corner of the screen with the coordinates (0,0). Recall from the previous chapter that the Mac screen is 512 pixels wide by 342 pixels high, and the default BASIC output window is about 491×254 . Thus (245,126) is the approximate middle of the screen, and the lower-right corner is (490,253).

To see how the LINE instruction works, enter this program:

```
LINE(0,0)-(490,253)
```

You can execute the program by selecting Start on the Run menu or by pressing COMMAND-R (the COMMAND key is on the bottom row of the keyboard, between OPTION and SPACE BAR).

LINE(0,0)-(490,253) draws a line from the upper-left corner of the window (0,0) to the lower-right corner (490,253). How you execute the program will determine which window is active after the program runs. If you use the Run menu, the List window reappears. If you use COMMAND-R, the Command window reappears.

Individual straight lines are not much of a challenge by them-

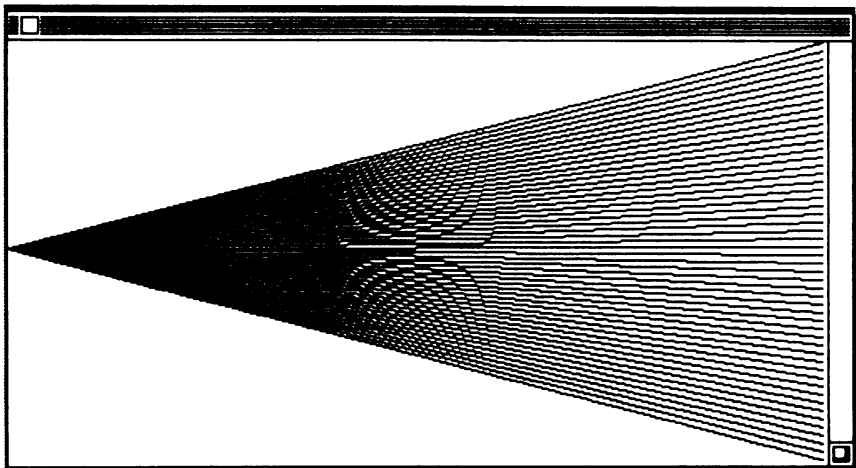


Figure 3-1.
Radial lines

selves, but in groups they can produce some pleasing patterns, such as those in Figure 3-1. Change your program to:

```
FOR Y=0 TO 255 STEP 5
  LINE(0,126)-(490,Y)
NEXT
stay: IF INKEY$="" THEN stay
```

This program draws a series of straight lines emanating from the point (0,126). If you look closely at the figure, you can see that each “straight line” on the Macintosh is actually a series of connected line segments. This segmentation creates an interesting interference pattern where the lines are close together.

The height of the right endpoint of each line ranges from the top of the window (0) to just below the bottom of the window (255) in steps of 5, so this pattern has 52 lines. Try different step increments to see how they will affect the figure and the pattern.

POLYGONS

Now that you know how to create straight lines, you can use them to build basic shapes. One such shape is a polygon, which is simply a figure with many straight sides, as in a triangle, a hexagon, or a rectangle.

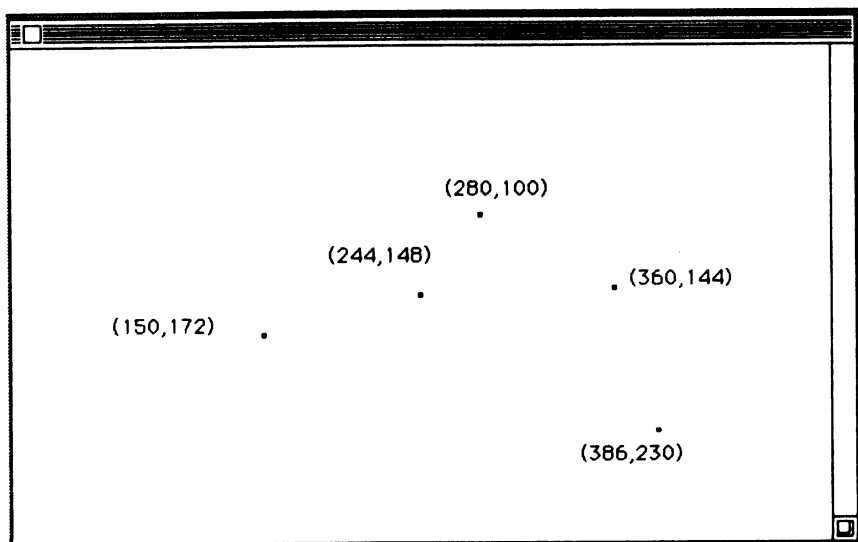
There are two typical approaches to drawing a polygon. One is to select a set of points and then join them with lines, as shown in Figure 3-2. The following program listing draws the figure.

```
x0=150: y0=172
x1=244: y1=148: GOSUB draw
x1=280: y1=100: GOSUB draw
x1=360: y1=144: GOSUB draw
x1=386: y1=230: GOSUB draw
x1=150: y1=172: GOSUB draw
stay: IF INKEY$="" THEN stay
STOP

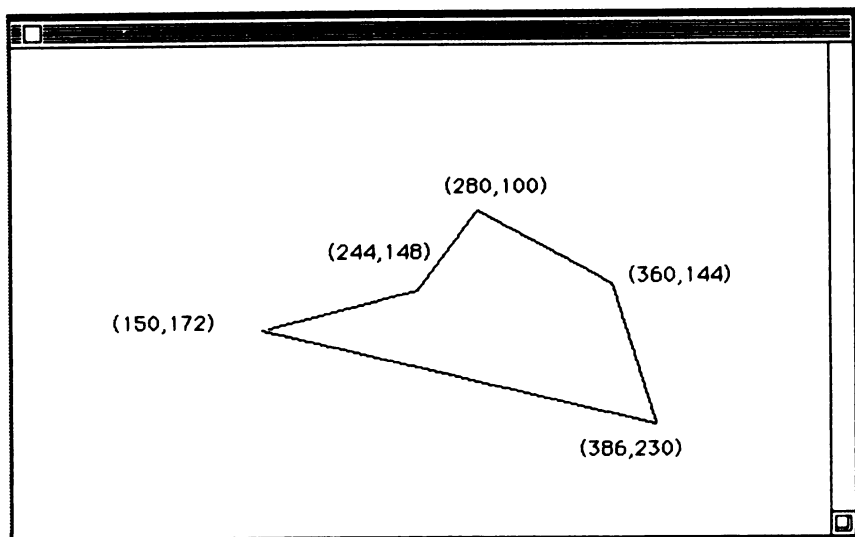
draw:
  LINE(x1,y1)-(x0,y0)
  x0=x1: y0=y1
RETURN
```

The program defines a pair of points (x0,y0) and (x1,y1) and calls a subroutine. This routine connects the points with a line and then redefines the starting point for the next line (x0,y0) as the endpoint of the current line (x1,y1).

Another way to draw a polygon is to define the endpoint of each



a.



b.

Figure 3-2.
Drawing a polygon by selecting points (a) and connecting them with lines (b)

line segment relative to the starting point. In the next program, each line starts at point (x0,y0); horizontal and vertical increments (h and v) are then added to get the endpoint (x1,y1).

```

RANDOMIZE TIMER
start:
CLS
x=245: y=126
x0=x: y0=y
FOR n=1 TO 5*RND+3
  h=200*RND-100: v=100*RND-50
  x1=x0+h: y1=y0+v
  LINE(x0,y0)-(x1,y1)
  x0=x1: y0=y1
NEXT n
LINE(x0,y0)-(x,y)
INPUT "Press [Enter] for next figure, [Control .] to stop";X$
GOTO start

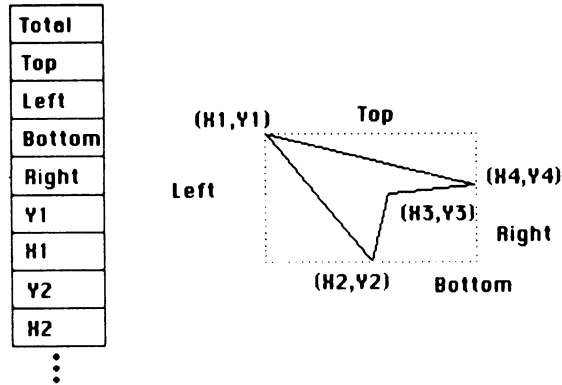
```

This program uses the built-in random number generator (discussed in Chapter 2) to draw random figures. `RANDOMIZE TIMER` ensures that each program run will be different from other program runs by grabbing a seed number from the system clock. `RND` selects a random number between 0 and 1. The `RND` function is used to select between three and eight sides for the figure and to pick random values for h and v. The line "`x1=x0+h: y1=y0+v`" calculates the next endpoint (x1,y1) by adding the values h and v to the previous endpoint (x0,y0).

Actually, lines in a true polygon do not cross each other. The purists among you may consider shapes like the one in Figure 3-3 to be two or more connecting polygons.

Polygons are such useful figures that there is a special Quick-Draw routine to simplify their creation. The setup is a bit tricky, so pay close attention. To draw a polygon, the x and y coordinates for the endpoints of each line segment must be stored in an integer array—an array with cells containing integers (see the discussion of arrays in Chapter 2).

In addition to the x and y coordinates, the first cell of the array must hold the number of bytes contained in the entire array. For an integer array, this number is twice the total number of cells in the array. The next four cells must contain the top, left, bottom, and right coordinates of a boundary rectangle that contains the entire polygon. The rest of the cells contain the endpoint coordinates in the unusual order y (vertical) followed by x (horizontal).



The following program demonstrates one way to prepare the data array for a polygon call.

```

DEFINT a-z: DIM p(30)
FOR i=0 TO 30: READ p(i): NEXT i
DATA 62,20,100,155,200,20,190,20,100,90,130,155,100,155
DATA 190,140,200,150,190,150,110,90,140,25,110,25,190
DATA 35,200,20,190
FRAMEPOLY VARPTR(p(0))
stay: IF INKEY$="" THEN stay

```

The first number (62) represents the total number of bytes in the array. The next four numbers specify the boundaries of the framing

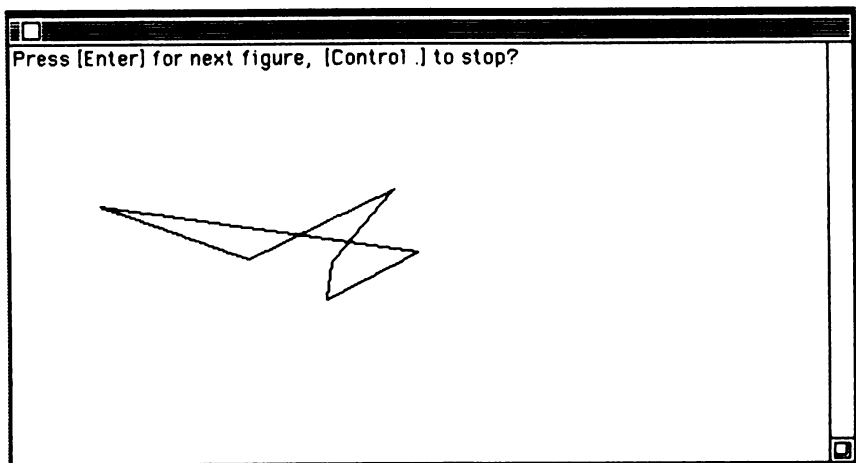


Figure 3-3.
Connecting polygons

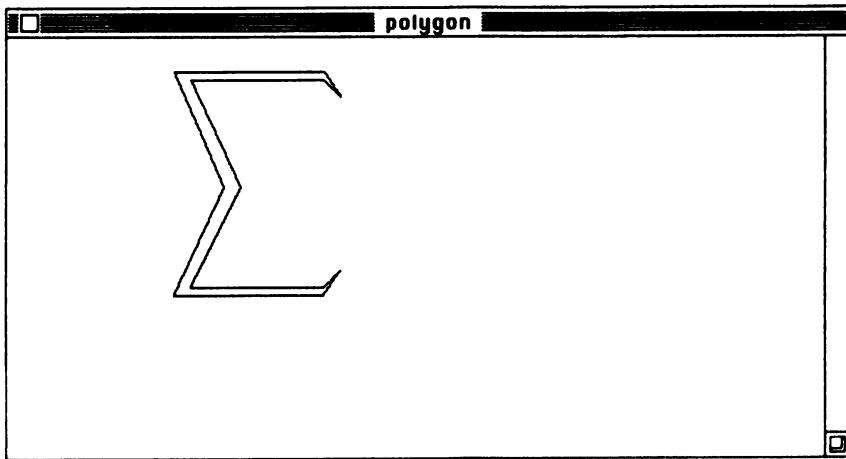


Figure 3-4.
Polygon drawn with FRAMEPOLY

rectangle. The rest of the numbers are coordinate pairs starting from the top right corner of the figure and proceeding in a counterclockwise direction (see Figure 3-4).

RECTANGLES

One of the most common polygonal shapes is the familiar rectangle. It can be drawn by connecting four straight lines so that opposite sides are parallel; but the Macintosh also provides an easier way to draw it. Just add the `b` option at the end of the `LINE` statement as shown below:

```
LINE(20,20)-(80,50),,b  
stay: IF INKEY$="" THEN stay
```

The two points (20,20) and (80,50) determine the upper-left and lower-right corners of the rectangle. (It doesn't matter which of these points you enter first.) The `b` option stands for box; it causes the computer to draw a rectangle instead of a line. The two commas mark the place of the missing color operand that you will see shortly.

ROUNDED RECTANGLES

A slight variation on the rectangle theme is a rectangle with rounded corners. These figures are used so often for organization charts and flow diagrams that the Mac designers included a special routine to

create them. Microsoft BASIC doesn't provide a separate statement for drawing rounded rectangles, so you'll have to call on the built-in ROM routines.

Before you start this exercise, take a brief look at the shapes available through calls to the ROM: arc (ARC), oval (OVAL), rectangle (RECT), rounded rectangle (ROUNDRECT), and polygon (POLY). Each of these shapes can be manipulated by the ERASE, FILL, FRAME, INVERT, and PAINT operations. You'll use each of these operations shortly. For now we'll focus on the FRAME operation that draws the outline of the specified shape.

It requires a bit more planning to draw shapes with the ROM routines than with BASIC statements, but being able to fill shapes with patterns and invert sections of the screen makes it worth extra effort. After a few gentle encounters, you'll include the ROM routines as permanent parts of your programming tools.

The format for the framed, rounded rectangle routine is

```
FRAMEROUNDRECT VARPTR(Rectangle%(0)),Ovalwidth, Ovalheight
```

Rectangle%(0) through Rectangle%(3) are four cells in an integer array that store the top, left, bottom, and right boundaries of the rectangle. VARPTR is the variable pointer function that returns the memory location of cell 0 in the Rectangle% array. Ovalwidth and Ovalheight store the width and height of the oval shape that will be used in rounding the corners. Recall from Chapter 2 that neither the CALL statement nor the associated parentheses are used for ROM calls in this book.

The boundaries of the rectangle deserve special comment because they are delivered to the routine in such a roundabout way. First, the numbers must be stored in an integer array. You can specify an integer variable with the DEFINT statement or by including the percent sign (%) in the variable name—A%(n), rect%(2), and so forth. The array should be a single-dimension list; only four consecutive array cells are required. Next, you deliver the address of the first of these cells to the routine with the variable pointer function, VARPTR. For example, if the numbers are stored in the array cells rect%(5), rect%(6), rect%(7), and rect%(8), then VARPTR(rect%(5)) tells the computer the actual memory address of the array cell rect%(5). Because array cells are stored consecutively in memory, the computer can find rect%(6) through rect%(8) on its own.

The Ovalwidth and Ovalheight parameters tell the computer the

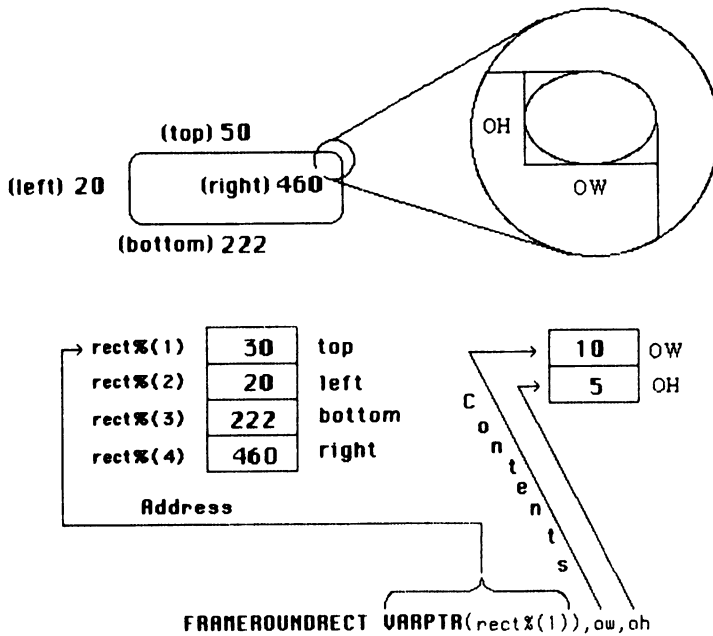


Figure 3-5.
FRAMEROUNRECT unveiled

width and height of the oval to be used in rounding the corners (Figure 3-5). The larger the numbers for the oval width(ow) and oval height(oh), the more the corners are rounded.

The following program draws several rounded rectangles (shown in Figure 3-6) with the same boundaries, but with different amounts of rounding at the corners:

```
FOR n=1 TO 4
  READ rect%(n)
  DATA 30,30,222,460
NEXT n
FOR d=0 TO 150 STEP 50
  ow=10+d: oh=5+d
  FRAMEROUNRECT VARPTR(rect%(1)),ow,oh
NEXT d
stay: IF INKEY$="" THEN stay
```

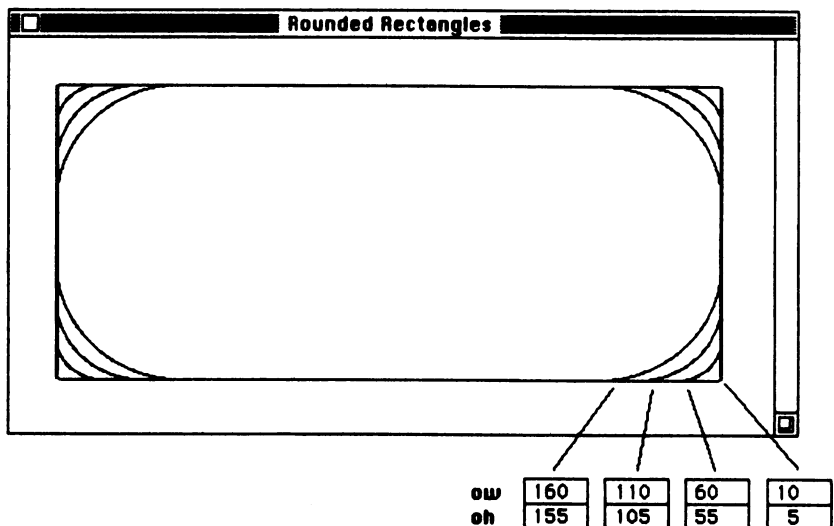


Figure 3-6.
Rounded rectangles

The first FOR/NEXT loop stores the data numbers for the top, left, bottom, and right edges of the figure into the variables rect%(1) through rect%(4). The FRAMEROUNRECT call does the actual drawing. VARPTR(rect%(1)) gives the address of the variable cell rect%(1) to the computer; ow specifies the width of the oval; and oh specifies the height of the oval. The FRAMEROUNRECT routine is used four times in the second FOR/NEXT loop to show four different rounding sizes.

CIRCLES

Circles, ovals, and arcs can all be drawn with the BASIC CIRCLE statement. To draw a simple circle, the computer requires a center point and radius. Here's how it's done:

```
FOR n=1 TO 4
  FOR radius = 1 TO 50 STEP n
    CIRCLE(110*n-30,120),radius
  NEXT radius
NEXT n
stay: IF INKEY$="" THEN stay
```

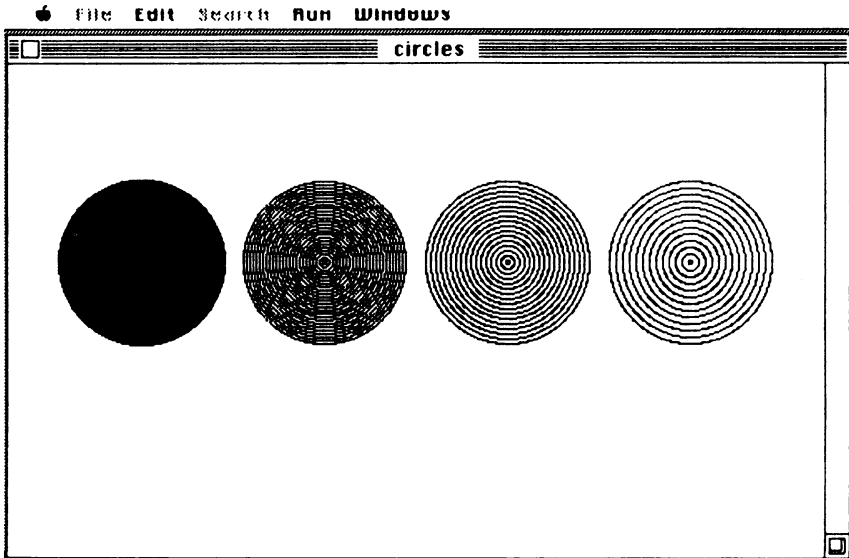


Figure 3-7.
Concentric circles

This program creates several concentric circles around each of four points. The radius increases at different rates around each of the center points (see Figure 3-7).

The **CIRCLE** statement also lets you include a **STEP** option, which is convenient for positioning the center of the circle relative to the current coordinates of the pen. Just as it does with **PSET**, including the word **STEP** changes the meaning of the **x,y** pair in the **CIRCLE** statement. In the following listing, **STEP(x,y)** determines the center of the circle by adding **x** and **y** to the coordinates of the last center:

```
RANDOMIZE TIMER
MOVETO 245,126
loop:
  x=4-8*RND(1): y=4-8*RND
  FOR n=1 TO 3+8*RND
    CIRCLE STEP(x,y),30
  NEXT n
IF INKEY$="" THEN loop
```

The **x** and **y** values set the direction and distance of the step. The **FOR/NEXT** loop repeats the **STEP** operation from 3 to 11 times,

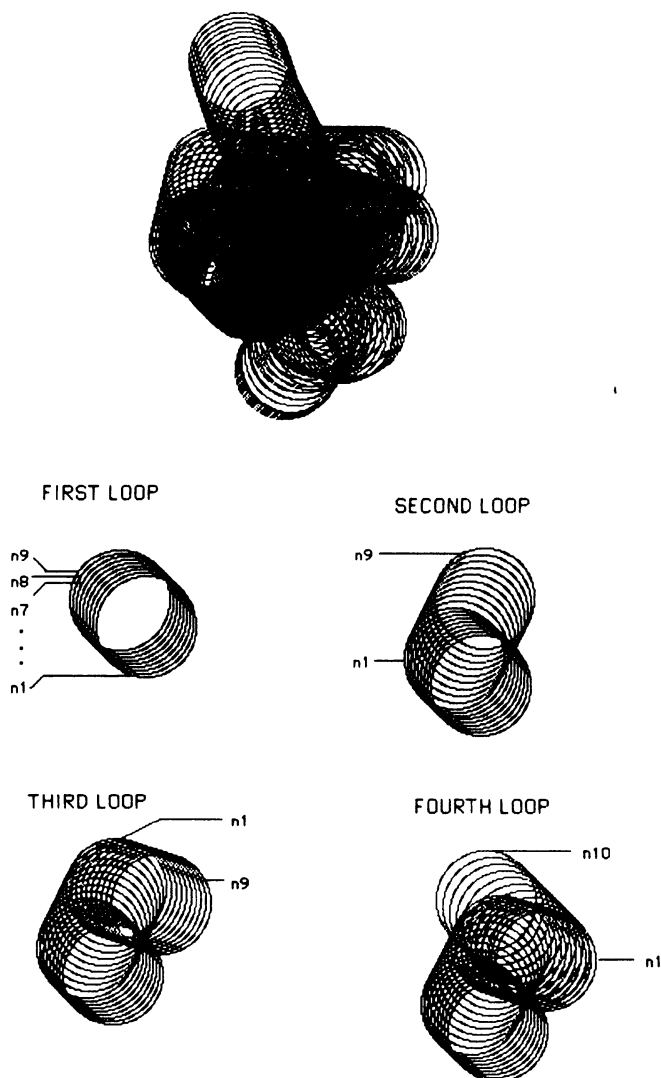


Figure 3-8.
Circles offset with STEP option

(3+8*RND), drawing a circle at each new location, as you can see in Figure 3-8.

OVALS

The **CIRCLE** statement is more versatile than its name implies—it can also be used to draw ovals. The **CIRCLE** statement has an optional parameter that specifies the aspect ratio of the oval (that is, the ratio of the x radius to the y radius). When the two radii are equal, their ratio is 1, and the figure is a round circle. Aspect ratios decreasing from 1 down to 0 define progressively flatter and flatter circles, approaching a horizontal line for the number 0. Numbers from 2 up define tall, skinny ovals that approach a straight vertical line. Figure 3-9 compares aspect ratios of 1, 0.5, and 5. Notice that the radius is used for the longest axis in all three cases.

The **CIRCLE** statement has several optional parameters. The complete format is

CIRCLE STEP(x,y),radius,color,start,end,aspect

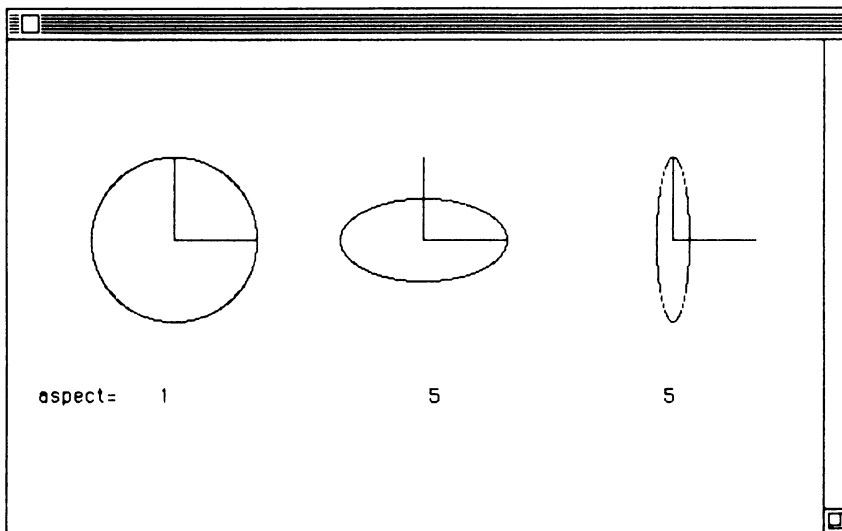


Figure 3-9.
Aspect ratios for ovals

Commas must be used as place holders for parameters that are not used, as shown in the following listing:

```
x=100: y=120: aspect=1: GOSUB oval
x=250: y=120: aspect=51: GOSUB oval
x=400: y=120: aspect=5: GOSUB oval
LOCATE 14,1
PRINT TAB(3)"aspect=";TAB(12)"1";TAB(32)".5";TAB(50)"5"
STOP
```

```
oval:
  CIRCLE(x,y),50,,,aspect
  LINE(x,y)-(x+50,y)
  LINE(x,y)-(x,y-50)
  RETURN
```

ARCS

What about the other optional parameters? The first one is color. It operates just like the color parameter in PSET and LINE; that is, it draws white if color is 30 and black if color is 33.

The start and end parameters let you draw only part of a circle or oval to create an arc. The two numbers specify the starting and ending angles of the arc. Angles are measured in a counterclockwise direction in radians, with 0 set at 3 o'clock. Those who slept through geometry class (and could care less about radians) only need to know what numbers to use to get different angles. The numbers range from 0 to 2π , where π is approximately equal to 3.14. Figure 3-10 shows angles for different multiples of π .

If you prefer working in degrees, you can convert from degrees to radians with a statement like $\text{rad} = \text{degree}/57.3$, because one radian equals approximately 57.3 degrees.

Negative values for the start or end numbers in a CIRCLE statement do not represent angles measured in the negative (clockwise) direction as you might have expected. Instead, a negative value causes the computer to draw a line from that endpoint of the arc to the center of the circle or oval. This is very useful for drawing pie charts with sections removed, as shown in Figure 3-11.

Enter the following program:

```
x=245: y=126: radius= 60: pi=3.141
CIRCLE(x,y),radius,,-.31*pi,-1.7*pi
CIRCLE(x+20,y),radius,-1.7*pi,-.31*pi
stay: IF INKEY$="" THEN stay
```

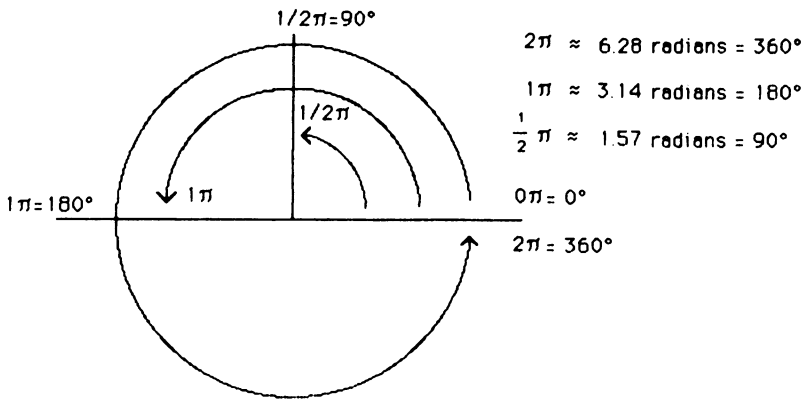


Figure 3-10.
Arc angles

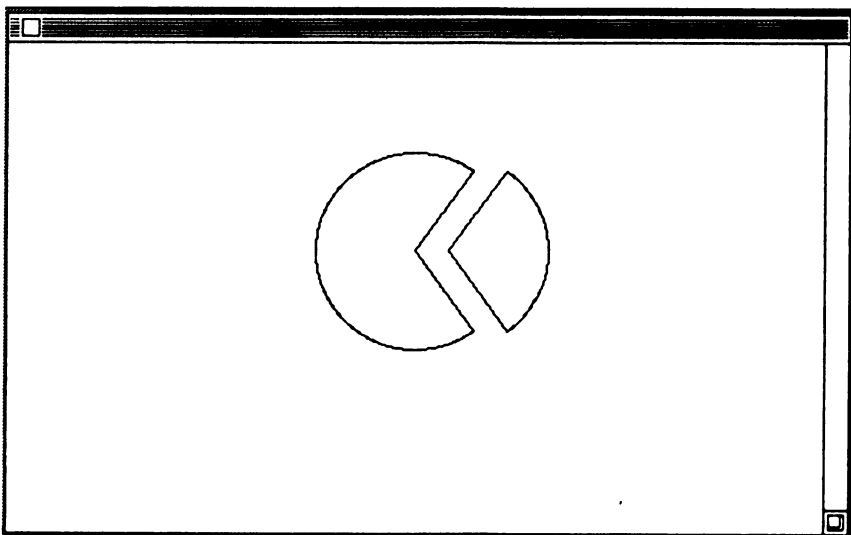


Figure 3-11.
Pie chart with section removed

Notice that the CIRCLE statement always measures angles (even negative ones) in a counterclockwise direction. The starting angle may be smaller or larger than the ending angle. In this program, the first CIRCLE statement draws the larger pie section from 0.3π to 1.7π . The second CIRCLE statement draws the smaller section from 1.7π to 0.3π . Try the program without the negative signs to see what happens.

Filling Shapes

Until now, the shapes you have drawn have been outlines of various figures. This section shows you how to fill in figures with different patterns.

The LINE statement has an optional parameter to let you fill in a rectangle. Just add an f after the trailing b; no comma is necessary, as shown in the following listing:

```
LINE(10,200)-(500,200)
FOR i=10 TO 500 STEP 50
  LINE(i,200)-(i,0)
  IF i=110 OR i=310 THEN skip
  LINE(i+35,0)-(i+65,125),,bf
skip: NEXT i
stay: IF INKEY$="" THEN stay
```

Figure 3-12 shows the output.

A filled rectangle can also be used to form a black background, so that you can draw white on black for a change of pace. But how do you draw white lines? The LINE and CIRCLE statements both have an optional parameter for color. The complete statement formats are shown below:

```
CIRCLE STEP(x,y),radius,color,start,end,aspect
LINE STEP(x1,y1)-STEP(x2,y2),color,BF
```

The number 30 has been assigned to white; 33 gives you the default color, black. Other numbers select either black or white on the Mac.

When you add bf to the end of the LINE statement, BASIC fills the rectangle with the color designated by the color parameter. The BASIC CIRCLE statement has no fill option. Try the following program:

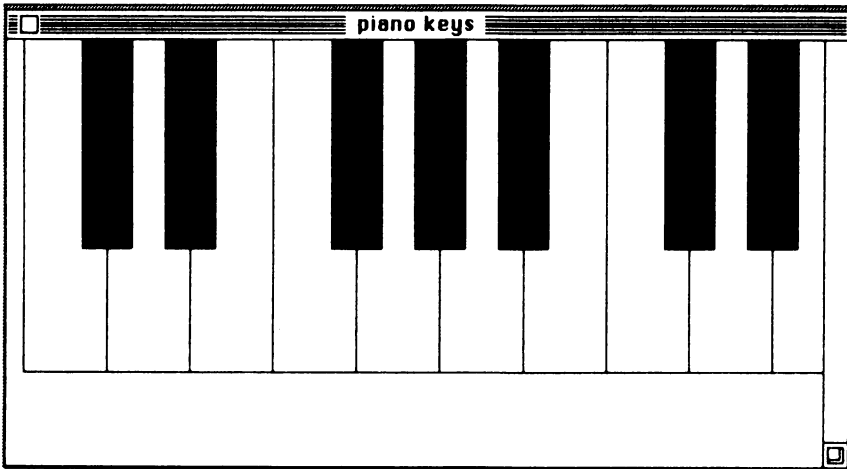


Figure 3-12.
Piano keys drawn with the LINE statement

```

LINE(0,0)-(490,253),,bf
CIRCLE(120,80),50,30
CIRCLE(240,80),50,30
CIRCLE(360,80),50,30
CIRCLE(180,120),50,30
CIRCLE(300,120),50,30
LINE(110,200)-(374,240),30,bf
MOVETO 160,230
TEXTSIZE 18
PRINT"XXIIIrd Olympiad";
TEXTSIZE 12
stay: IF INKEY$="" THEN stay

```

The first LINE statement fills the output window with the default color, black. The five circles are drawn in white (color=30). The second LINE statement creates a white rectangle. Notice that the fill option (f) fills in the selected color (30=white). The next four lines position the text, print it, and return the text size back to 12-point, as you see in Figure 3-13.

BLACK OR WHITE WITH ROM ROUTINES

You can fill other shapes with either black or white as well, using the built-in ROM routines. You can either PAINT or ERASE each of the

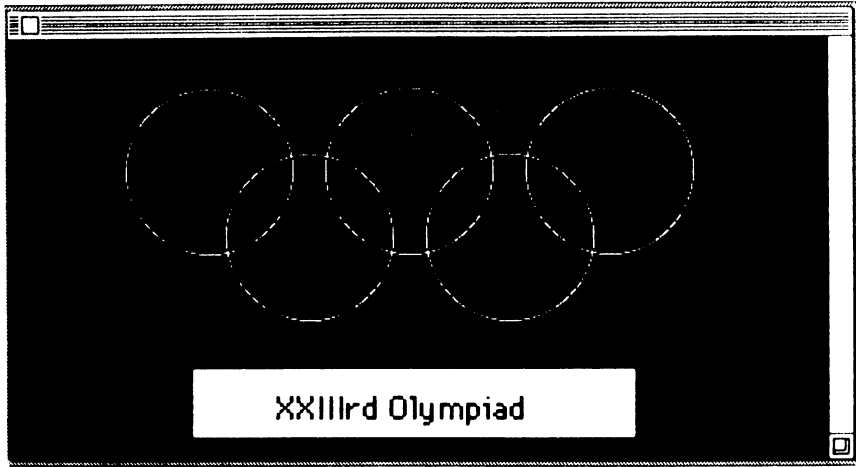
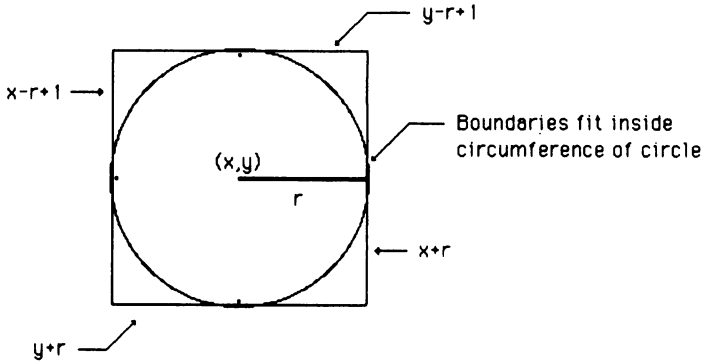


Figure 3-13.
White circles on black

five shapes: rectangle, rounded rectangle, oval, arc, or polygon. Painting or erasing a shape means filling that shape with a pattern similar to those used in MacPaint. PAINT fills a selected shape with the current pen pattern; the default pattern is black. ERASE fills a shape with the current background pattern; the default pattern is white. (You will see how to change these default patterns in the section below. For now, keep their default colors, black and white.)

Start with the ERASE operation. You'll use this operation to "white out" the interiors of the circles that were shown in Figure 3-8. This exercise illustrates how differently BASIC statements and ROM routines define shapes. The BASIC CIRCLE statement defines a circle by its center and radius. ROM calls require the top, left, bottom, and right boundaries of the smallest square that can contain the circle. The translation is fairly straightforward. For a circle with center (x,y) and radius r, the boundaries are

$$\begin{aligned}\text{Top} &= y - r + 1 \\ \text{Left} &= x - r + 1 \\ \text{Bottom} &= y + r \\ \text{Right} &= x + r\end{aligned}$$



An extra "+ 1" has been added to the top and left numbers so that the erased area does not include the circle itself. There is no need to subtract 1 from the bottom and right boundaries, because there are really two different coordinate systems involved here. BASIC statements refer to screen pixel locations; ROM routines refer to grid lines between the pixel locations. Figure 3-14 shows the difference.

The top, bottom, left, and right numbers refer to infinitely thin grid lines *between* screen pixels. The circle coordinates refer to actual pixel locations. As you can see in Figure 3-14, the grid numbers required to frame the inside of a figure are not exactly the same as

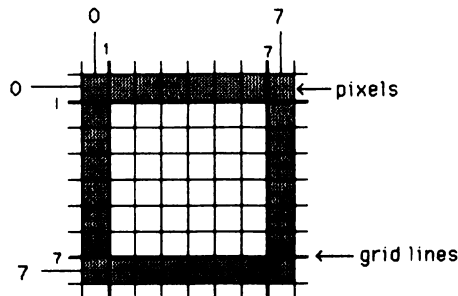


Figure 3-14.
Grid line coordinates versus pixel coordinates

those of the border pixels themselves. The top and left grid numbers are one greater; the bottom and right grid numbers are the same.

ERASEOVAL uses VARPTR to find the location of the array containing the boundaries, just as FRAMEROUNRECT used VARPTR in Figure 3-5.

Now you are ready to use the ERASEOVAL routine. Type this:

```
RANDOMIZE TIMER
cx=245: cy=126: radius=30
MOVETO cx,cy
loop:
  x=4-8*RND(1): y=4-8*RND(1)
  FOR n=1 TO 3+8*RND(1)
    cx=cx+x: cy=cy+y
    CIRCLE (cx,cy),radius
    array%(0)=cy-radius+1: array%(1)=cx-radius+1
    array%(2)=cy+radius: array%(3)=cx+radius
    ERASEOVAL VARPTR(array%(0))
  NEXT n
IF INKEY$="" THEN loop
```

This listing generates Figure 3-15. Compare the output to Figure 3-8 to see the difference made by erasing the center of each circle.

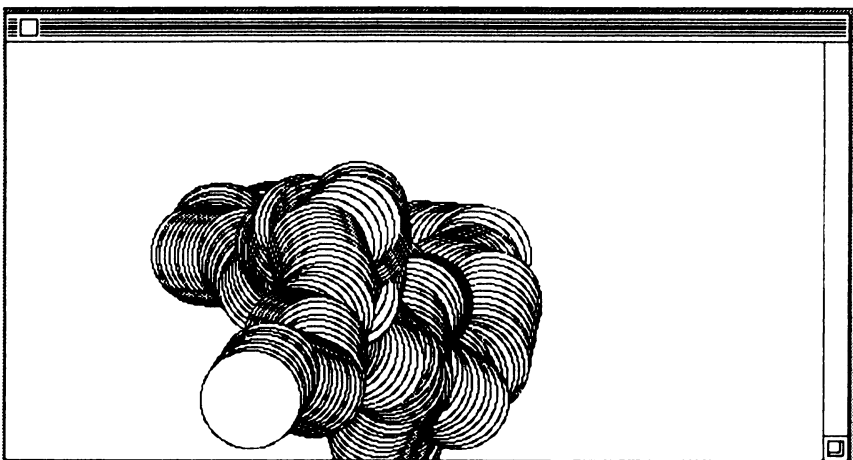


Figure 3-15.
ERASEOVAL exercise

MIXING ROM ROUTINES AND BASIC STATEMENTS

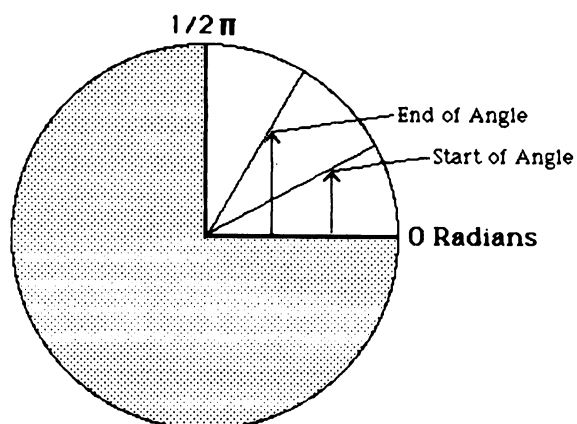
Using ROM calls will greatly increase your productivity, but mixing figures drawn by these calls with figures drawn by BASIC statements can be a challenge. Filling in arcs—for example, when a pie chart is drawn with BASIC statements and needs to be filled using ROM calls—requires special attention. BASIC statements measure angles in radians, whereas ROM calls use degrees. To make matters worse, BASIC statements place zero at 3 o'clock and measure angles counterclockwise; ROM calls place zero at 12 o'clock and measure angles clockwise, as shown in Figure 3-16.

There can be other difficulties also. If the rectangle you used to frame your arc for the ROM call was not a perfect square, the PAINT-ARC routine will use the corners of the rectangle as 45-degree marks, as shown in Figure 3-17. Angles will therefore be stretched and contracted to conform to the rectangular shape. If you try to mix and match the BASIC CIRCLE statement and the ROM ARC routine, you should use a square to frame the arc, or you will have quite a job matching the two.

To make translating angle measures between ROM calls (degrees) and BASIC statements (radians) easier, here are two programs to do the work for you. The first listing converts from radians to degrees; the second converts from degrees to radians. Both assume that the boundary rectangle for all arc ROM calls is a square.

```
PRINT "Enter radian angles between 0 and 6.28 (2 $\pi$ )"
INPUT "Input the start of the arc in radians";rad1
start=450-rad1*180/3.1416!
s1: IF start>360 THEN start=start-360: GOTO s1
s2: IF start<0 THEN start=start+360: GOTO s2
INPUT "Input the end of the arc in radians";rad2
endarc=450-rad2*180/3.1416!
e1: IF endarc>360 THEN endarc=endarc-360: GOTO e1
e2: IF endarc<0 THEN endarc=endarc+360: GOTO e2
arcangle=ABS(endarc-start)
PRINT "Input direction: (Type a '-' for clockwise, ";
INPUT "or a '+' for counterclockwise ";dir$
IF dir$="-" AND endarc<start THEN arcangle=360-arcangle
IF dir$="+" AND endarc>start THEN arcangle=360-arcangle
IF arcangle>359 THEN arcangle=arcangle-360
IF dir$="+" THEN arcangle=-arcangle
PRINT "Start angle" CINT(start)
PRINT "End of angle" CINT(endarc)
PRINT "Arc of angle" CINT(arcangle)
PRINT "Continue (y/n)?";
```

HOW MICROSOFT BASIC STATEMENTS MEASURE ANGLES



HOW MACINTOSH ROM ROUTINES MEASURE ANGLES

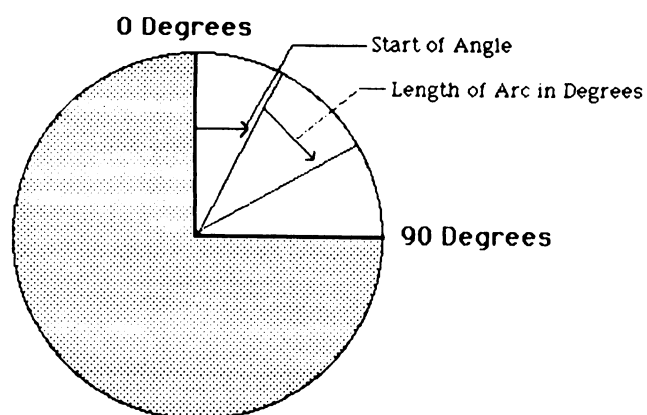


Figure 3-16.
Different ways of measuring angles

```

done:
more$=INKEY$
IF more$="-n" OR more$="-N" THEN STOP
IF more$="-y" OR more$="-Y" THEN RUN ELSE done

PRINT "Input the start of the angle in degrees (0-360)";
INPUT start
pi=3.14159!
sr=7.854!-pi/180*start
IF sr < 0 THEN sr=sr+2*pi
IF sr > 6.2832! THEN sr=sr-2*pi
PRINT "Input the arc angle in degrees"
INPUT "pos = clockwise, neg = counterclockwise";angle
endarc=7.854!-pi/180*(start+angle)
IF endarc<=0 THEN endarc=endarc+6.2832!
IF endarc>6.2832! THEN endarc=endarc-6.2832!
arc=7.854!-pi/180*ABS(angle)
IF arc>6.2832! THEN arc=arc-6.2832!
PRINT "the start of the arc in rads is " sr
PRINT "the end of the arc in rads is " endarc
PRINT "the arc is " arc
PRINT "Continue (y/n)?";
done:
more$=INKEY$
IF more$="-n" OR more$="-N" THEN STOP
IF more$="-y" OR more$="-Y" THEN RUN ELSE done

```

The next program uses the first conversion routine, along with the PAINT operation, to paint the pie slice in Figure 3-11 black.

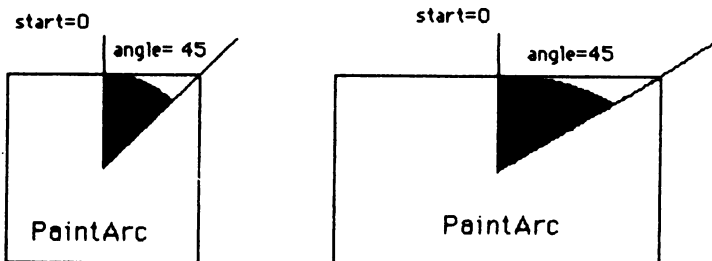


Figure 3-17.
Measuring 45° angles on squares and on rectangles that are not square

```

x=240 : y=150 : radius= 60: pi=3.1416!
CIRCLE(x,y),radius,,-.3!*pi,-1.7!*pi
CIRCLE(x+20,y),radius,, -1.7!*pi,-.3!*pi
rad1=1.7!*pi: rad2=.3!*pi: d$=""
GOSUB ConvertToDegree
Array%(0)=y-radius: Array%(1)=x+20-radius
Array%(2)=y+radius: Array%(3)=x+20+radius
PAINTARC VARPTR(Array%(0)),start,arcangle
stay: IF INKEY$="" THEN stay
END

```

```

ConvertToDegree:
start=450-rad1*180/3.1416!
s1: IF start>=360 THEN start=start-360: GOTO s1
s2: IF start<0 THEN start=start+360: GOTO s2
endarc=450-rad2*180/3.1416!
e1: IF endarc>=360 THEN endarc=endarc-360: GOTO e1
e2: IF endarc<0 THEN endarc=endarc+360: GOTO e2
arcangle=ABS(endarc-start)
IF d$="--" AND endarc<start THEN arcangle=360-arcangle
IF d$="+" AND endarc>start THEN arcangle=360-arcangle
IF arcangle>359 THEN arcangle=arcangle-360
IF d$="+" THEN arcangle=-arcangle
RETURN

```

This program takes the starting and ending angles of the pie slice (1.7π and 0.3π) and the direction (counterclockwise) and converts them into a start angle and an arc angle in degrees, as required by the PAINTARC routine. Figure 3-18 shows the result.

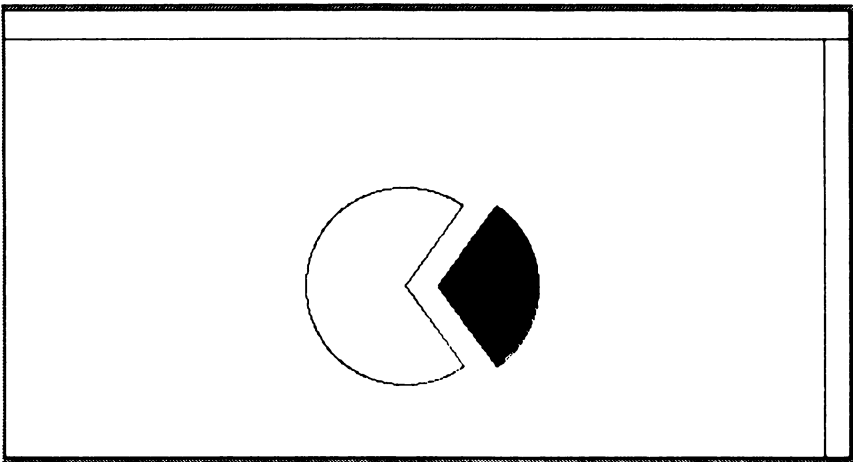


Figure 3-18.
Painted arc

The Mac has one more trick up its sleeve to aid you in changing colors of different shapes. An INVERT operation can be used to change the color (black/white) of any of four basic Mac shapes—rectangles, rounded rectangles, ovals, and arcs. The operation turns each white dot black and each black dot white in the selected area, similar to the way a photograph and its negative are related. You can use this operation to invert the title of Figure 3-13.

```

LINE(0,0)-(500,300),,bf
CIRCLE(120,80),50,30
CIRCLE(240,80),50,30
CIRCLE(360,80),50,30
CIRCLE(180,120),50,30
CIRCLE(300,120),50,30
LINE(110,200)-(374,240),30,bf
MOVETO 160,230
TEXTSIZE 18
PRINT"XXIIIrd Olympiad";
TEXTSIZE 12
array%(1)=201: array%(2)=111
array%(3)=240: array%(4)=374

loop:
  FOR delay=1 TO 3000: NEXT delay
  INVERTRECT VARPTR(array%(1))
  IF INKEY$="" THEN loop

```

Press COMMAND-. to stop the program. Figure 3-19 shows the inverted title.

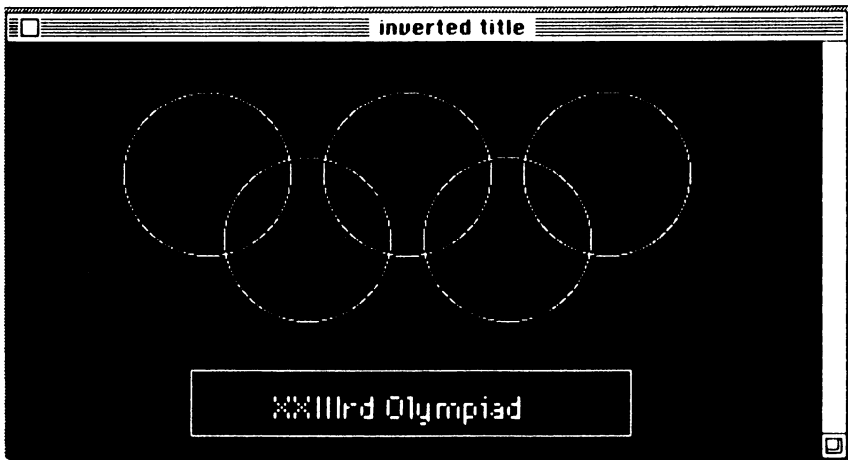


Figure 3-19.
Inverted title

DEFINING YOUR OWN PATTERNS

The PAINT and ERASE operations let you fill a variety of shapes with the default pen pattern and background patterns, which are initially set to white and black respectively. With the FILL call, you can fill shapes with your own patterns, just as you can in MacPaint. Again, you have to use the ROM calls. Here's how it works.

Each pattern is composed of 64 dots arranged in 8 rows and 8 columns. The Macintosh expects you to design a pattern and then store it in four consecutive cells in an integer array such as pattern%(1), pattern%(2), pattern%(3), and pattern%(4). The percent sign is included to make the variable an integer type (instead of single- or double-precision), which is critical to this process. Figure 3-20 shows how to translate a sample pattern into the four numbers that will be stored in an array.

The 8×8 matrix is divided into four sections, each corresponding to an array cell. To calculate the number for each cell, add the values of the pixels you choose. For example, pixels labeled 4096, 512, and 4 were selected in the top section. The sum 5512 is stored in array cell p%(0). Note that white (no pixels selected) is represented by 0, and black (all cells selected) is represented by 1 + 2 + 4 + . . . + 8192 + 16384 = 32768, which equals -1.

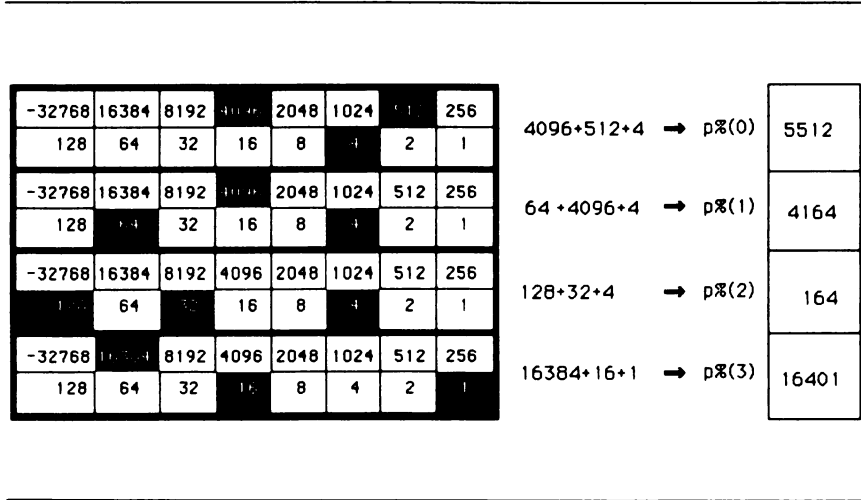


Figure 3-20.
Translating a pattern to four numbers

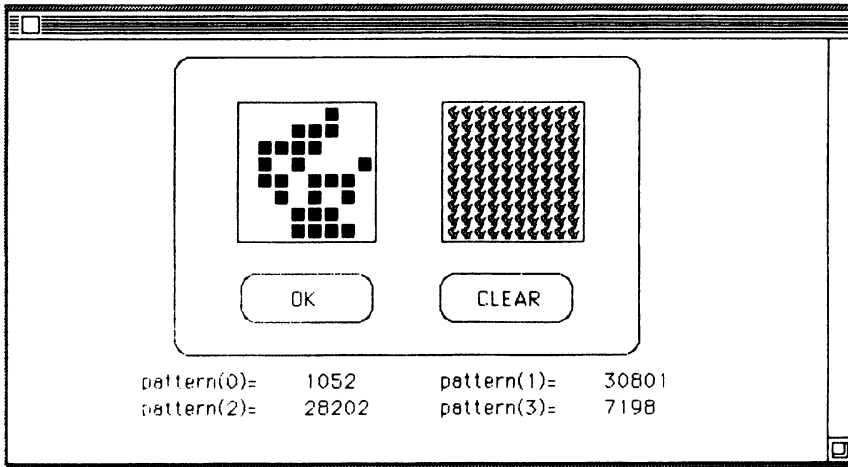


Figure 3-21.
Pattern design program

Laying out an 8×8 grid and calculating each of these numbers every time you want to create a new pattern would be a chore. To speed up the process, here's a utility program that does all the work for you. Figure 3-21 shows the sample output. Here is the listing:

```
DEFINT a-z: DEFSGN x
DIM pat(8,8),pattern(3),powers(8,2)
start:
FOR row=1 TO 8
  FOR col=1 TO 8
    pat(row,col)=0
  NEXT col
NEXT row
x=1
FOR row=2 TO 1 STEP -1
  FOR col=8 TO 1 STEP -1
    IF row=1 AND col=1 THEN powers(col,row)=-32768!
    ELSE powers(col,row)=x: x=2*x
  NEXT col
NEXT row
prec(0)=40: prec(1)=264
prec(2)=120: prec(3)=344
CLS
MOVETO 170,160: PRINT "OK";
MOVETO 282,160: PRINT "CLEAR";
FOR n=0 TO 2
  FOR k=0 TO 3
    READ rect(k)
  NEXT k
```

```

FRAMEROUNRECT VARPTR(rect(0)),20,20
NEXT n
DATA 10,100,190,380,140,140,170,220,140,260,170,340
LINE(138,37)-(221,121),,b
LINE(261,37)-(346,121),,b
MOVETO 80,210
PRINT "pattern(0)=";PTAB(260);"pattern(1)="
PRINT PTAB(80);"pattern(2)=";PTAB(260);"pattern(3)="

CheckMouse:
IF MOUSE(0)<>-1 THEN CheckMouse
x=MOUSE(1): y=MOUSE(2)
IF ABS(180-x)>39 OR ABS(80-y)>39 THEN boxes
x=INT(x/10): col=x-13
y=INT(y/10): row=y-3
color=pat(col,row)

ChangePattern:
pat(col,row)=1-pat(col,row)
IF pat(col,row)=1 THEN LINE(x*10+1,y*10+1)-(x*10+8,y*10+8),,bf
ELSE LINE(x*10+1,y*10+1)-(x*10+8,y*10+8),30,bf
IF row MOD(2)=0 THEN rx=2! ELSE rx=1
IF pat(col,row)=1 THEN pattern(INT((row-1)/2))=pattern
(INT((row-1)/2))+powers(col,rx)
IF pat(col,row)=0 THEN pattern(INT((row-1)/2))=pattern
(INT((row-1)/2))-powers(col,rx)
FILLRECT VARPTR(prec(0)),VARPTR(pattern(0))
MOVETO 170,210: PRINT pattern(0);
MOVETO 350,210: PRINT pattern(1);
MOVETO 170,226: PRINT pattern(2);
MOVETO 350,226: PRINT pattern(3);

mouseloop:
IF MOUSE(0)>=0 THEN CheckMouse
x=MOUSE(1): y=MOUSE(2)
IF ABS(180-x)>39 OR ABS(80-y)>39 THEN mouseloop
x=INT(x/10): col=x-13
y=INT(y/10): row=y-3
IF color=pat(col,row) THEN ChangePattern
GOTO mouseloop

boxes:
IF ABS(180-MOUSE(1))<40 AND ABS(155-MOUSE(2))<15 THEN STOP
IF ABS(300-MOUSE(1))<40 AND ABS(155-MOUSE(2))<15 THEN RUN
GOTO CheckMouse

```

Advanced programmers may wish to examine the inner workings of this program; the rest of you can simply type it in and use it.

Remember, as we mentioned just after the Introduction, long programs like this one are available on disk. If you decide to look the program over, don't be dismayed by the mouse statements used in the CheckMouse and mousetloop sections of the program. These statements will be covered in detail in the next chapter.

The program works like the Edit Pattern feature of MacPaint; you set or reset dots by clicking or dragging the mouse. The resulting pattern is displayed in the window on the right, and the pattern numbers are shown below the windows. Clicking the Clear button erases the current pattern and restarts the program. Clicking the OK button stops the program. This program can be used by itself as a stand-alone utility, or it can be incorporated into programs that make heavy use of patterns.

What do you do with the four pattern numbers once they have been determined? These numbers should be stored in a single-dimension array (see the discussion in Chapter 2) and then accessed by the FILL operation with one of these basic shapes: RECT, ROUNDRECT, OVAL, ARC, and POLY. The numbers can also be used with PENPAT and BACKPAT to change the default pen and background patterns. To show you how this FILL operation works, the next program fills the empty pie section in Figure 3-18 with a pattern.

```

DIM pat%(11)
x=240 : y=150 : radius= 60: pi=3.1416
CIRCLE(x,y),radius,,-.3*pi,-1.7*pi
CIRCLE(x+20,y),radius,,-1.7*pi,-2.3*pi
rad1=1.7*pi: rad2=2.3*pi: d$=""
GOSUB ConvertToDegree:
Array%(0)=y-radius: Array%(1)=x+20-radius
Array%(2)=y+radius: Array%(3)=x+20+radius
PAINTARC VARPTR(Array%(0)),start,arcangle
rad1=.3*pi: rad2=1.7*pi: d$=""
GOSUB ConvertToDegree:
Array%(0)=((y-radius)+1): Array%(1)=((x-radius)+1)
Array%(2)=y+radius: Array%(3)=x+radius
FOR i%=0 TO 11
  READ pat% (i%)
NEXT i%
DATA -32446,9240,6180,17025,27647,-18945,-8581,-8329
DATA -32512,24,6144,129
FILLARC VARPTR(Array%(0)),start,arcangle,VARPTR(pat%(0))
stay: IF INKEY$="" THEN stay
END

```

```

ConvertToDegree:
start=450-rad1*180/3.1416
s1: IF start>=360 THEN start=start-360: GOTO s1
s2: IF start<0 THEN start=start+360: GOTO s2
endarc=450-rad2*180/3.1416
e1: IF endarc>=360 THEN endarc=endarc-360: GOTO e1
e2: IF endarc<0 THEN endarc=endarc+360: GOTO e2
arcangle=ABS(endarc-start)
IF d$="-"AND endarc<start THEN arcangle=360-arcangle
IF d$="+"AND endarc>start THEN arcangle=360-arcangle
IF arcangle>359 THEN arcangle=arcangle-360
IF d$="+" THEN arcangle=-arcangle
RETURN

```

The pattern is stored in the array `pat%`. The conversion subroutine at the end of the program converts the radian angles from the `CIRCLE` statements to degrees for both sections. Figure 3-22 shows the result.

Applications

Your new-found ability to draw shapes and patterns with only a few key statements enables you to do fairly sophisticated graphics with minimal effort. The rest of this chapter gives you some ideas to work with.

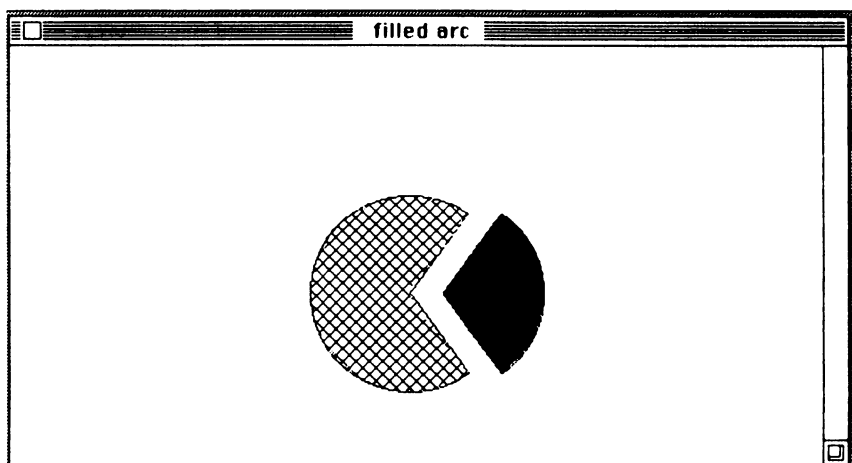


Figure 3-22.
Arc filled with a pattern

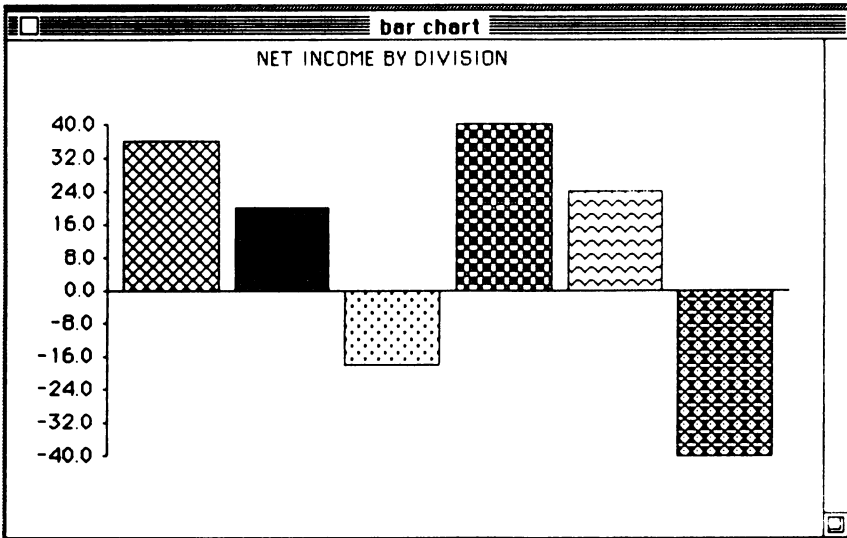


Figure 3-23.
Bar chart

BAR CHART

The next program showcases many of the techniques you learned in this chapter. It accepts a set of numbers and displays them in bar-chart form. The program is designed to accommodate as many as 15 numbers, both positive and negative. Figure 3-23 shows the output from a sample run.

The program illustrates how the industrious programmer can toss a customized graphics program together with minimal effort. Here is the listing:

```
DEFINT e,w,c,r,p
DIM pat(60), rect(60), dat(20), h(20)
FOR i=0 TO 59
  READ pat(i)
NEXT i
DATA -32446,9240,6180,17025,27647,-18945,-8581,-8329
DATA -32512,24,6144,129,-7262,-7396,5148,-7262
DATA -31932,10256,0,0,6204,32385,-32386,15385
DATA 4096,129,0,16,774,3096,14460,-332
DATA 16,14460,14352,0,-1,-20071,-29307,-1,32,0,0,0
DATA 871,-4388,-26618,3612,-32704,9240,6180,513
DATA 16,4152,31868,14352,-30396,8738,-188,-30446
n=0
```

```

entry:
n=n+1
CLS
INPUT "Enter data. When done, press Enter";num$
dat(n)=VAL(num$)
IF num$<>" " GOTO entry
IF n=1 THEN n=0: GOTO entry
n=n-1: dmax=dat(1): dmin=dat(1)

draw:
CLS
FOR i=1 TO n
  IF dat(i)<dmin THEN dmin=dat(i)
  IF dat(i)>dmax THEN dmax=dat(i)
NEXT i
IF dmax<0 THEN dmax=0
IF dmin>0 THEN dmin=0
l=dmax
FOR i=50 TO 250 STEP 20
  MOVETO 10,i+4
  PRINT USING "#####.0-";l,
  l=l-(dmax-dmin)/10
  LINE(58,i)-(60,i)
NEXT i
LINE(60,50)-(60,250)
baseline=dmax/(dmax-dmin)*200+50
IF dmin>0 THEN baseline=250
FOR i=1 TO n
  h(i)=baseline-dat(i)/(dmax-dmin)*200
NEXT i
LINE(60,baseline)-(470,baseline)
FOR i=1 TO n
  LINE(70+(400/n)*(i-1),h(i))-(70+(400/n)*i-10,baseline),,b
  rect(4*(i-1))=h(i)+1
  IF dat(i)<0 THEN rect(4*(i-1))=baseline+1
  rect(4*(i-1)+1)=70+(400/n)*(i-1)+1
  rect(4*(i-1)+2)=baseline
  IF dat(i)<0 THEN rect(4*(i-1)+2)=h(i)
  rect(4*(i-1)+3)=70+(400/n)*i-10
NEXT i
MOVETO 150,15
PRINT "NET INCOME BY DIVISION"
FOR n=0 TO 56 STEP 4
  FILLRECT VARPTR(rect(n)),VARPTR(pat(n))
NEXT n
stay: IF INKEY$="" THEN stay

```

The vertical axis contains 11 tick marks. The minimum and maximum are calculated from the numbers entered, and the zero line is adjusted accordingly. Fifteen patterns are read into the array pat in

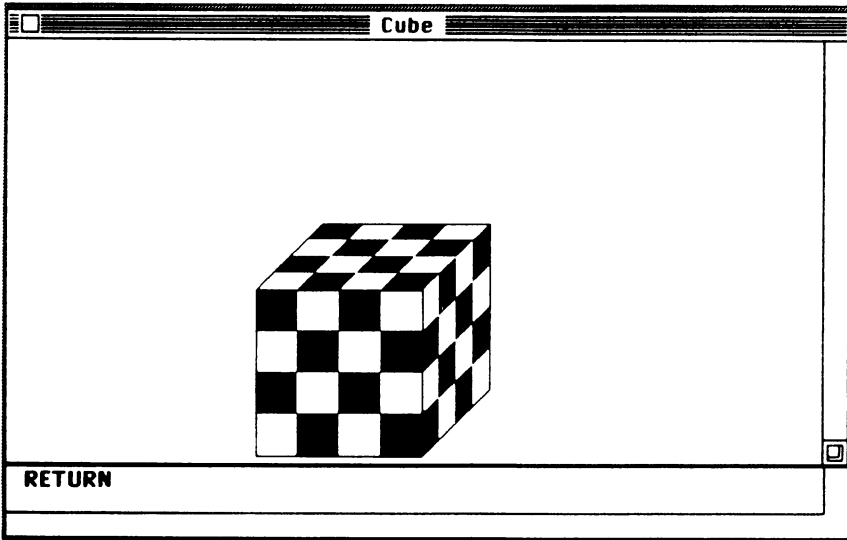


Figure 3-24.
Cube

locations 0 through 59, with four array cells per pattern. The border numbers for each rectangle are stored in `rect`, and `FILLRECT` fills each rectangle with a different pattern.

The nice thing about having this program in BASIC is that you can modify it as needed. You can add labels, read data from disk files, and so forth.

CUBE

To illustrate the Mac's ability to handle polygons, the following program draws three faces for a cube. Looking at Figure 3-24, you might assume that `FILLRECT` could be used to fill in all the black areas; but the "squares" on the top and right faces of the cube are not really squares at all. They are polygons.

The program listing is straightforward:

```
DEFINT a-z
DIM p(23),q(13),r(13),r1(4),z(13)

'Start polygon for the front
FOR j=0 TO 3: READ r1(j): NEXT j
DATA 150,150,175,175
```

'Load coordinates for the frame of the polygon

FOR i=0 TO 22: READ p(i): NEXT i

DATA 46,110,150,250,290,150,150,250,150,250,250,150,250

DATA 150,150,110,190,110,290,210,290,250,250

'Coordinates for the starting polygon for the top

FOR i=0 TO 12: READ q(i): NEXT i

DATA 26,140,175,150,210,150,175,150,200,140,210,140,185

'Coordinates for the starting polygon for the side

FOR i=0 TO 12: READ r(i): NEXT i

DATA 26,165,250,200,260,175,250,200,250,190,260,165,260

FRAMEPOLY VARPTR (p(0))

GOSUB FRONT

GOSUB TOP

GOSUB SIDE

stay: IF INKEY\$="" THEN stay

END

FRONT:

FOR k=1 TO 2

FOR i=1 TO 2

PAINTRECT VARPTR(r1(0))

FOR j=0 TO 3: Q1(j)=r1(j): NEXT j

Q1(0)=Q1(0) + 50 : Q1(2)=Q1(2) + 50

PAINTRECT VARPTR(Q1(0))

r1(1)=r1(1) + 50 : r1(3)=r1(3) + 50

NEXT i

r1(0)=175: r1(1)=175 : r1(2)=200 : r1(3)=200

NEXT k

RETURN

TOP:

FOR k=1 TO 2

FOR j=1 TO 2

PAINTPOLY VARPTR(q(0))

FOR i=0 TO 12: z(i)=q(i): NEXT i

FOR i=2 TO 12 STEP 2: z(i) = z(i) + 50: NEXT i

PAINTPOLY VARPTR(z(0))

FOR i=1 TO 11 STEP 2: q(i) = q(i) - 20: NEXT i

FOR i=2 TO 12 STEP 2: q(i) = q(i) + 20: NEXT i

NEXT j

FOR i=1 TO 11 STEP 2: q(i) = q(i) + 30: NEXT i

FOR i=2 TO 12 STEP 2: q(i) = q(i) - 55: NEXT i

NEXT k

RETURN

SIDE:

FOR k=1 TO 2

FOR j=1 TO 2

PAINTPOLY VARPTR(r(0))

FOR i=0 TO 12: z (i)=r(i): NEXT i

```

FOR i=1 TO 11 STEP 2: z(i)=z(i) + 50: NEXT i
PAINTPOLY VARPTR(z(o))
FOR i=1 TO 11 STEP 2: r(i)=r(i) - 20: NEXT i
FOR i=2 TO 12 STEP 2: r(i)=r(i) + 20: NEXT i
NEXT j
FOR i=1 TO 11 STEP 2: r(i)=r(i) + 5: NEXT i
FOR i=2 TO 12 STEP 2: r(i)=r(i) - 30: NEXT i
NEXT k
RETURN

```

The first portion loads arrays with the appropriate coordinates. Array *p* holds the endpoints for all the edges of the cube. Arrays *r1*, *q*, and *r* contain the coordinates for a single polygon on a particular face. The coordinates for the other polygons are calculated by manipulating these numbers with the subroutines *FRONT*, *TOP*, and *SIDE*.

VIEWSCREEN

This program is a modification of the *viewscreen* program in Chapter 2. The *LINE* statement is used to black out the screen, and the *CIRCLE* statement is used to erase and paint stars. The larger stars are created with three concentric circles, each with a radius of 0, 1, and 2 respectively (see Figure 3-25).

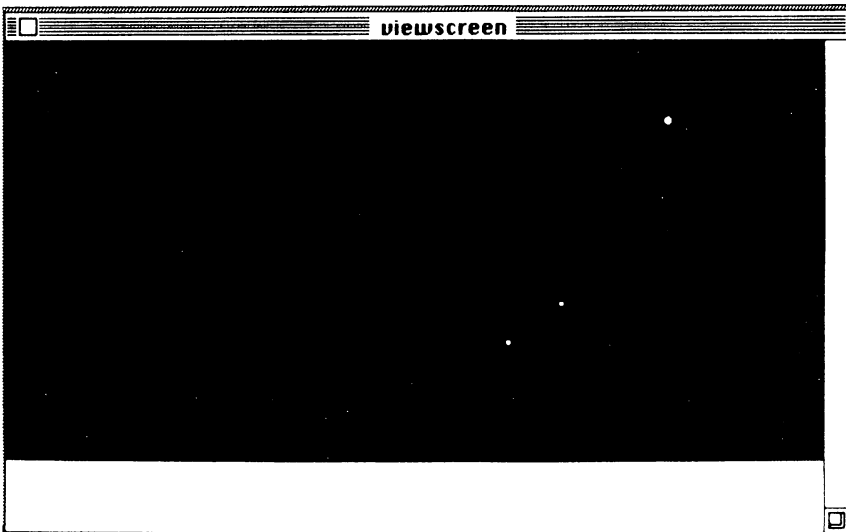


Figure 3-25.
Revised *viewscreen* (from Chapter 2)

```

DEFINT a-z: DEFMSG d: RANDOMIZE TIMER
DIM b(30,4)
c(0)=125: c(1)=244: c(2)=127: c(3)=246
n=4
FOR i=1 TO n: r(i)=1: NEXT i
LINE (0,0)-(491,254),33,bf
FOR i=1 TO 100: PRESET(490*RND,253*RND): NEXT i
a:
FOR i=1 TO n
  CIRCLE (x(i),y(i)),0,33
  IF r(i)>0 THEN CIRCLE (x(i),y(i)),1,33
  IF r(i)>1 THEN CIRCLE (x(i),y(i)),2,33
  IF x(i)+h(i)>0 AND x(i)+h(i)<490 AND y(i)+k(i)>0 AND
y(i)+k(i)<253 THEN c
  f(i)=0
  b:
  x(i)=203+84*RND
  h(i)=(x(i)-245)/5: IF h(i)=0 THEN b
  y(i)=108+36*RND
  k(i)=(y(i)-126)/5: IF k(i)=0 THEN b ELSE GOTO d
  c:
  h(i)=1.4*h(i): k(i)=1.4*k(i)
  x(i)=x(i)+h(i): y(i)=y(i)+k(i)
  d=(x(i)-245)^2+(y(i)-126)^2
  IF d>20000 THEN f(i)=2: GOTO d
  IF d>6000 THEN f(i)=1
  d:
  CIRCLE (x(i),y(i)),0,30
  IF f(i)>0 THEN CIRCLE (x(i),y(i)),1,30
  IF f(i)>1 THEN CIRCLE (x(i),y(i)),2,30
NEXT i
PSET(RND*490,253*RND),RND
stay: IF INKEY$="" THEN a

```

One of the improvements in this version of the program is that the stars accelerate as they get closer to the viewer. The program line after label c increases the size of the step to encourage the illusion of increased speed.

In order to increase the general activity level, the PSET statement at the end of the program adds an occasional star to the background.

FUNCTION PLOT

The function plot program in Chapter 2 brought up the question of how to maintain a continuous curve with steep vertical lines when plotting functions. The LINE statement you learned in this chapter can help you with the problem. Try the following:

```

TEXTSIZE 10
TEXTFONT 10
TEXTMODE 1
LINE(50,130)-(450,130)
FOR x=50 TO 450 STEP 20
  IF x=250 THEN skip
  LINE (x,128)-(x,132)
  left=10: IF x=50 OR x=450 THEN left=15
  MOVETO x-left,144: PRINT (x-50)/20-10;
skip: NEXT x
FOR y=10 TO 250 STEP 20
  IF y=130 THEN MOVETO 240,135: GOTO jump
  MOVETO 232,y+5
  jump:
  PRINT 5-(y-30)/20;
  LINE (248,y)-(252,y)
NEXT y
LINE (250,10)-(250,250)
FOR x=-10 TO 10 STEP .011
   $y = 2 * (x * x - 9 * x + 8) / (2 * x * x + x - 4)$ 
   $x1 = 20 * x + 250$ 
   $y1 = -20 * y + 130$ 
  IF y>6 OR y<-6 THEN jump1
  IF x=-10 THEN PSET (x1,y1) ELSE LINE (x0,y0)-(x1,y1)
  jump1:
  x0=x1: y0=y1
NEXT x
TEXTSIZE 12
TEXTMODE 0
TEXTFONT 1
stay: IF INKEY$="" THEN stay

```

This program deals with the problem by drawing lines between consecutive graph-point pairs, instead of plotting them one at a time. Using new variables (x0,y0) to store the previous point coordinates, you can draw a line connecting (x0,y0) to (x1,y1) for all points except the first. Figure 3-26 shows the results.

LIFE

The LINE statement really speeds up the drawing of the grid in the program called Life, which was introduced in Chapter 2. Another major change in the program is the addition of a new array, nb, which stores the number of neighbors for each cell. With this information stored in an array, the number of neighbors for each cell can be calculated during the initial board setup and updated as needed during the updateboard section. With these changes, the program operates much faster.

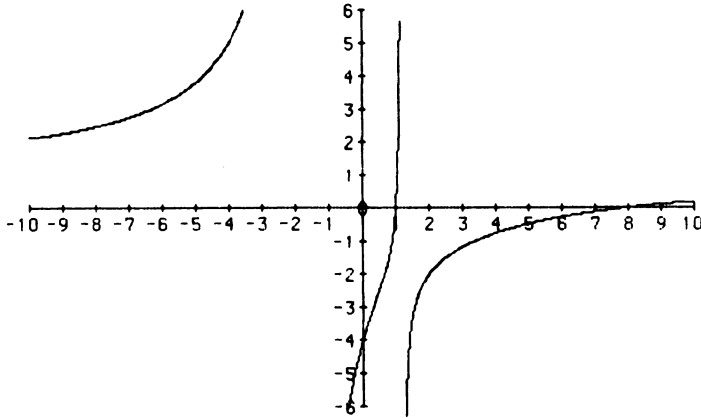


Figure 3-26.
Function plot with LINE statement

```

DEFINT a-z: DIM a(25,25), nb(25,25): g=1: p=0
PRINT "initializing game board: generation: ";g;
FOR r=1 TO 25: READ s$
  FOR c=1 TO 25
    a(r,c)=VAL(MID$(s$,c,1))
    IF a(r,c)=0 THEN exit3
    LINE(170+8*c,45+8*r)-(172+8*c,47+8*r),bf: p=p+1
  FOR h=-1 TO +1
    IF c+h<1 OR c+h>25 THEN exit2
    FOR v=-1 TO +1
      IF r+v<1 OR r+v>25 OR h=0 AND v=0 THEN exit1:
      nb(r+v,c+h)=nb(r+v,c+h)+1
    exit1:
  NEXT v
exit2:
NEXT h
exit3:
NEXT c
NEXT r
PRINT "population":p
FOR x=175 TO 375 STEP 8
  LINE (x,50)-(x,250)
NEXT x
FOR y=50 TO 250 STEP 8
  LINE (175,y)-(375,y)

```

```

NEXT y
LOCATE 1,1: PRINT SPACES$(80)
LOCATE 1,1: PRINT "generation";g;" population";p

checkstatus:
p=0
FOR r=1 TO 25
  LOCATE 2,1: PRINT"Scanning for neighbors:row";r
  FOR c=1 TO 25: nb=0: IF a(r,c)=1 THEN p=p+1
    IF a(r,c)=1 THEN IF nb(r,c)<2 OR nb(r,c)>3 THEN a(r,c)=4 ELSE a(r,c)=2
    IF a(r,c)=0 THEN IF nb(r,c)=3 THEN a(r,c)=3
  NEXT c
NEXT r

updateboard:
g=g+1
LOCATE 1,1: PRINT "generation";g;" population";p
FOR r=1 TO 25
  FOR c=1 TO 25: f=0
    IF a(r,c)<3 THEN exit6
    bw=33: IF a(r,c)=4 THEN bw=30
    LINE (170+8*c,45+8*r)-(172+8*c,47+8*r),bw,bf
    FOR h=-1 TO +1
      IF c+h<1 OR c+h>25 THEN exit5
      FOR v=-1 TO +1
        IF r+v<1 OR r+v>25 OR h=0 AND v=0 THEN exit4
        IF bw=33 THEN nb(r+v,c+h)=nb(r+v,c+h)+1 ELSE nb(r+v,c+h)=nb(r+v,c+h)-1
      NEXT v
    NEXT h
  NEXT c
  exit6:
  exit5:
  exit4:
  exit6:
  NEXT c
NEXT r

redoarray:
p=0
FOR r=1 TO 25: FOR c=1 TO 25
  IF a(r,c)=2 OR a(r,c)=3 THEN a(r,c)=1
  IF a(r,c)=4 THEN a(r,c)=0
  IF a(r,c)=1 THEN p=p+1
NEXT c,r
GOTO CheckStatus:
DATA "10010000000000000000000000000000"
DATA "00001000000000000000000000000000"
DATA "10001000000000000000000000000000"
DATA "01111000000000000000000000000000"
DATA "00000000000000000000000000000000"
DATA "00000000000010000000000000000000"
DATA "00000000000010000000000000000000"
DATA "00000000000100000000000000000000"

```

```
DATA "00000000010000000000000000"
DATA "00000000100000000000000000"
DATA "00011111000000000000000000"
DATA "00011110000000000000000000"
DATA "00010110000000000000000000"
DATA "00000000000000000000000000"
DATA "00000000000000000000000000"
DATA "00000000100100000000000000"
DATA "00000000111100000000000000"
DATA "00000001000010000000000000"
DATA "00000001011010000000000000"
DATA "00000001000010000000000000"
DATA "00000000111100000000000000"
DATA "00000000000000000000000000"
DATA "00000000001000000000000000"
DATA "00000000000000000000000000"
DATA "00000000000000000000000000"
```

CREATING A LOGO

Here is another idea for a company logo, based on the techniques for drawing shapes that you have learned in this chapter. Figure 3-27 was drawn primarily by using the polygon ROM calls.

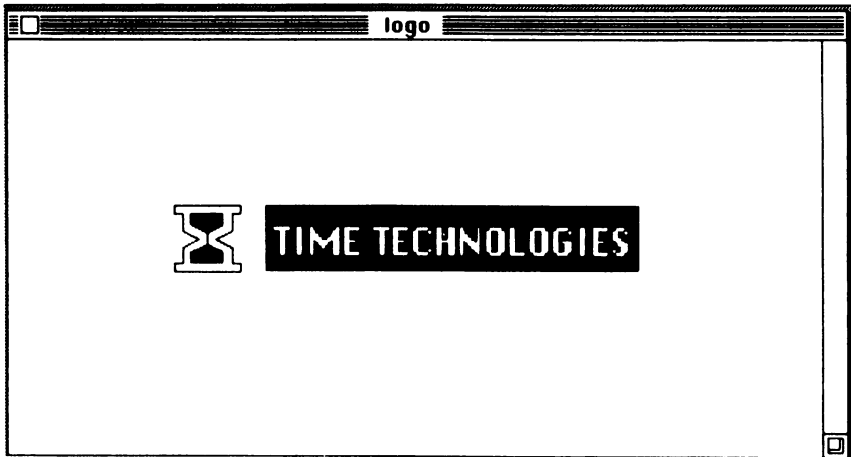


Figure 3-27.
Logo


```

DEFINT a-z
DIM p(43), q(17), r(17)
FOR i=0 TO 42: READ p(i): NEXT i
DATA 86,10,10,10,10,10,100,100,105,100,105,105,115,105
DATA 120,115,125,105, 135,105,135,100,140,100,140
DATA 140,135,140,135,135,125,135,120,125,115,135
DATA 105,135,105,140,100,140,100,100
FOR i=0 TO 16: READ q(i): NEXT i
DATA 34,123,110,135,130,128,110,135,110,135,130,128
DATA 130,123,120,128,110
FOR i=0 TO 16: READ r(i): NEXT i
DATA 34,105,110,117,130,105,110,112,110,117,120,112,130
DATA 105,130,105,110
FRAMEPOLY VARPTR(p(0))
PAINTPOLY VARPTR(q(0))
PAINTPOLY VARPTR(r(0))
LINE (120,117)-(120,123)
TEXTFONT 0: TEXTSIZE 22
MOVETO 160,130
PRINT "TIME TECHNOLOGIES"
TEXTFONT 1: TEXTSIZE 12
f(0)=100: f(1)=155: f(2)=140: f(3)=380
INVERTRECT VARPTR(f(0))
stay: IF INKEY$="" THEN stay

```

The array *p* contains the coordinates for the outside border. The boundaries of the black sections are stored in arrays *q* and *r*. The company name is printed in 22-point Chicago and then inverted with *INVERTRECT*.

Summary

You are now well on your way toward being able to control graphics on your Macintosh with BASIC. This chapter has shown you how you can draw and fill basic shapes, including lines, rectangles, circles, arcs, ovals, and polygons. The ROM routines also give you the ability to fill these shapes with patterns, paint them with the current pen pattern, erase them with the current background pattern, and invert them.

In the next chapter, you will learn how to make full use of the mouse to create interactive graphics programs.

BASIC Statements and Functions

CIRCLE STEP(x,y),radius,color,start,end,aspect	RETURN
GOSUB	STOP
INPUT	TAB
LINE STEP(x1,y1)–STEP(x2,y2),color,BF	VARPTR
PRINT USING	

ROM Shape Operations

ERASEshape	INVERTshape
FILLshape	PAINTshape
FRAMEshape	

ROM Shapes

RECT—rectangle	ARC—arc
ROUNDRECT—round rectangle	POLY—polygon
OVAL—oval	

4

Interactive Graphics

Graphics is no longer static, non-moving art. With the computer, artists can easily interact with their art as they create it. If they don't like a particular effect, they can change or remove it. Engineers can use computers interactively to design automobiles, ships, airplanes — almost anything you can name. Game enthusiasts can design and control game graphics in a way never before possible.

Traditional computer programs permitted only a stilted form of interaction. As the user, you would enter some data; then you could run the program. After the program ran, you could examine the results and note any errors. Only then could you change your data or your program and start over.

An interactive program, on the other hand, accepts data and

immediately shows results, which makes a computer less tedious to program and use. An interactive program also permits a computer to perform tasks that were previously impractical or impossible. Imagine, for example, trying to draw a picture by entering coordinates from the keyboard and looking at the results later!

This chapter presents techniques you can use to make your programs more interactive. You will learn to use the mouse, control the cursor's appearance, detect and act on interaction events, and create dialog boxes and buttons. The last part of the chapter presents five applications that use interactive programming. These include a maze, a personality trait analyzer, an interactive educational application, a cursor shape editor, and a puzzle.

Mouse Input

We usually interact with a computer through a keyboard. This is a convenient way to enter text, although we all await the day when we can simply dictate text to the machine. But for pointing and sketching, the keyboard leaves a lot to be desired. Tools for collecting input that are better than the keyboard for pointing activities include touch-sensitive screens, light pens, digitizing tablets, and mouse devices.

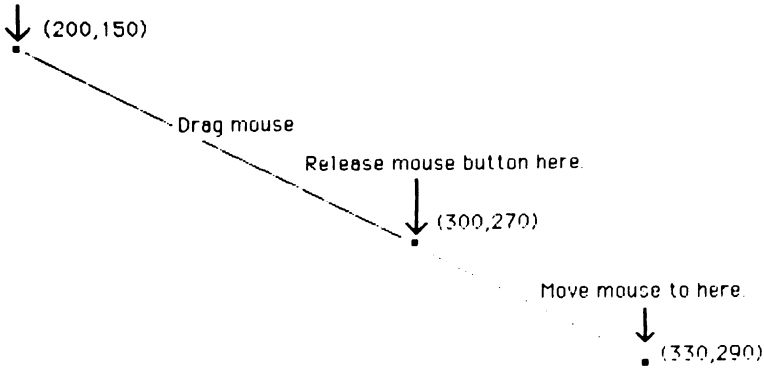
The Macintosh mouse is an excellent pointer for selecting menu options and objects on the screen. It can also be used to control a pen for drawing, although some artists prefer a stylus or pen. With practice, however, you can produce satisfactory drawings using a mouse.

As the mouse moves around the desktop, the motion of its roller ball changes the position of the arrow-shaped cursor on the screen. The position of the mouse and the status of the mouse button (down or up) can also be detected with the `MOUSE` function. Your programs can use the cursor position and mouse-button status to control many things. For example, when the button is down, you can draw lines between successive cursor positions to create a kind of electronic pen-and-ink drawing.

POLLING THE MOUSE

The Mac has an elaborate system for keeping track of the cursor position as well as the status of the mouse button. You can detect such things as the starting and ending positions of a drag, the number of button clicks since the last status check, and the current pointer position.

Press mouse button here.



Results of Calls.

<u>Function</u>	<u>Value</u>	<u>Description</u>
Mouse (0).	1	Button up. Single click since last mouse (0) call
Mouse (1).	330	Current x coordinate of cursor
Mouse (2).	290	Current y coordinate of cursor
Mouse (3).	200	Starting x coordinate of cursor
Mouse (4).	150	Starting y coordinate of cursor
Mouse (5).	300	Ending x coordinate of cursor
Mouse (6).	270	Ending y coordinate of cursor

Figure 4-1.
Status of the mouse after a drag

The key to discovering the status of the mouse is the MOUSE function. MOUSE is actually seven functions, labeled MOUSE(0) through MOUSE(6), that return specific information about the mouse (see Table 4-1). Figure 4-1 illustrates the status after a single drag operation.

Table 4-1.
MOUSE Function

MOUSE(0):	Button status
0:	Button inactive since last MOUSE(0) call
1:	Button up; single click since last MOUSE(0) call
2:	Button up; double click since last MOUSE(0) call
3:	Button up; triple click since last MOUSE(0) call
-1:	Button down; single click since last MOUSE(0) call
-2:	Button down; double click since last MOUSE(0) call
-3:	Button down; triple click since last MOUSE(0) call
MOUSE(1):	Current x coordinate of cursor
MOUSE(2):	Current y coordinate of cursor
MOUSE(3):	Starting x coordinate of cursor
MOUSE(4):	Starting y coordinate of cursor
MOUSE(5):	Ending x coordinate of cursor
MOUSE(6):	Ending y coordinate of cursor

In this scenario, the program user presses the button, drags the mouse, releases the button, and then moves the mouse a short distance. Then the program executes calls to MOUSE(0) through MOUSE(6). MOUSE(0) returns 1 because the button is up and a single click occurred. MOUSE(1) and MOUSE(2) return the current cursor position. MOUSE(3) and MOUSE(4) return the starting position of the drag. MOUSE(5) and MOUSE(6) return the ending position of the drag.

Your program can use all of this feedback, or only a selected portion. You can type a short example that tests if the button is clicked in a certain area. Enter

```

LINE (40,40)-(70,70),,b
TestClick:
IF MOUSE(0)=0 THEN TestClick
IF MOUSE(3)<40 OR MOUSE(3)>70 OR MOUSE(4)<40 OR MOUSE(4)>70
  THEN TestClick
MOVETO 94,62
PRINT "rectangle selected"
WHILE MOUSE(0)<>2: WEND

```

Try pressing the button anywhere outside the rectangle. You won't get a reaction—the program only responds to a click within the specified area. The second IF statement sends program control back to

the TestClick line if the button press occurs outside the outlined rectangle.

Notice that the mouse test used in this program [MOUSE(0)=0] is designed to detect *any* button activity. If the button has been pressed since the last MOUSE call, MOUSE(0) will be 1, 2, 3, -1, -2, or -3. If the button is not pressed, MOUSE(0) is 0, and control passes back to the TestClick line. Also, by choosing to test MOUSE(3) and MOUSE(4), the program requires that the cursor be inside the rectangle when the button is pressed, not when it is released.

ANOTHER EXAMPLE OF MOUSE INTERACTION

The different mouse functions can be confusing at first. The next program lets you see the values change right before your eyes as you click the button and move the mouse around the screen.

```
CLS : TEXTMODE 0
InitDisplay:
LOCATE 1,1
PRINT "Button Status", "Mouse(0)="MOUSE(0)" "
PRINT "Current X,Y", "Mouse(1)="MOUSE(1);TAB(32); "Mouse(2)="MOUSE(2)" "
PRINT "Starting X,Y", "Mouse(3)="MOUSE(3);TAB(32); "Mouse(4)="MOUSE(4)" "
PRINT "Ending X,Y", "Mouse(5)="MOUSE(5);TAB(32); "Mouse(6)="MOUSE(6)" ";
UpdateDisplay:
LOCATE 1,23: PRINT MOUSE(0)
LOCATE 2,23: PRINT MOUSE(1);TAB(40);MOUSE(2)
LOCATE 3,23: PRINT MOUSE(3);TAB(40);MOUSE(4)
LOCATE 4,23: PRINT MOUSE(5);TAB(40);MOUSE(6)
MOVETO MOUSE(1),MOUSE(2)
IF MOUSE(0)<0 THEN LINETO MOUSE(1),MOUSE(2)
GOTO UpdateDisplay
```

These simplified examples do not fully illustrate why you might like to determine when a click occurs in a selected area. One typical use is in menu selection. You can set up your own menu list and invite the user to select one of several options. The Mac has several built-in routines for doing this, and you will use them later in this chapter. Here is a do-it-yourself version:

```
DEFINT a-z:
TEXTFONT 1
TEXTSIZE 18
TEXTFACE 64
DIM pat(12),box(12),edge(12)
FOR i=0 TO 11: READ pat(i): NEXT i
```

```

FOR i=0 TO 11: READ box(i): NEXT i
LINE (40,40)-(70,70),,b
LINE (40,100)-(70,130),,b
LINE(40,160)-(70,190),,b
MOVETO 94,65
PRINT "Balance Sheet"
MOVETO 94,125
PRINT "General Ledger"
MOVETO 94,185
PRINT "Income Statement"
DATA 21930,21930,21930,21930,-1,-1,-1,-1
DATA 258,1032,4128,16512
DATA 41,41,70,70,101,41,130,70,161,41,190,70
FILLRECT VARPTR(box(0)),VARPTR(pat(0))
FILLRECT VARPTR(box(4)),VARPTR(pat(4))
FILLRECT VARPTR(box(8)),VARPTR(pat(8))
FOR i=0 TO 11: READ edge(i):NEXT i
DATA 40,90,70,244,100,90,130,255,169,90,190,285
TestClick:
IF MOUSE(0) =0 THEN TestClick
IF MOUSE(3)>40 AND MOUSE(3)<70 AND MOUSE(4)>40 AND
MOUSE(4)<70 THEN GOTO Invert1
IF MOUSE(3)>40 AND MOUSE(3)<70 AND MOUSE(4)>100 AND
MOUSE(4)<130 THEN GOTO Invert2
IF MOUSE(3)>40 AND MOUSE(3)<70 AND MOUSE(4)>160 AND
MOUSE(4)<190 THEN GOTO Invert3
GOTO TestClick
Invert1: INVERTRECT VARPTR(edge(0)): GOTO CleanUp
Invert2: INVERTRECT VARPTR(edge(4)): GOTO CleanUp
Invert3: INVERTRECT VARPTR(edge(8))
CleanUp:
TEXTFONT 3
TEXTSIZE 12
TEXTFACE 0
WHILE MOUSE(0)<>2: WEND

```

This program uses the INVERTRECT ROM routine to highlight the selected option. Its results are illustrated in Figure 4-2.

You can also use mouse clicking and dragging to manipulate objects on the screen interactively. The next example allows a user to draw rectangles by clicking a starting point and dragging to control the size of the rectangle. This program works like the rectangle routine in MacPaint.

```

DEFINT a-z
FOR i=1 TO 4: READ pat(i): NEXT i
DATA -21931,-21931,-21931,-21931
Loop:
WHILE MOUSE(0)=0
IF INKEY$<>" THEN STOP
WEND

```



```

x1←MOUSE(3): y1←MOUSE(4)
x2=x1: y2=y1
PENPAT VARPTR(pat(1))
PENMODE 10
WHILE MOUSE(0)≠0
  rec(1)=y1: rec(3)=y2
  IF y2<y1 THEN rec(1)=y2: rec(3)=y1
  rec(2)=x1: rec(4)=x2
  IF x2<x1 THEN rec(2)=x2: rec(4)=x1
  FRAMERECT VARPTR(rec(1))
  x3=x2: y3=y2
  WHILE (x3=x2 AND y3=y2)
    dummy←MOUSE(0)
    x3=MOUSE(1): y3=MOUSE(2)
  WEND
  FRAMERECT VARPTR(rec(1))
  x2=x3: y2=y3
WEND
PENNORMAL
FRAMERECT VARPTR(rec(1))
GOTO Loop

```

These are only a few of the ways in which polling the MOUSE function can be used to control program interaction. You probably have several uses in mind already, but just in case, you can explore many of the variations in this chapter and throughout the book. Next we'll take a look at the mouse's alter ego on the screen, the cursor.

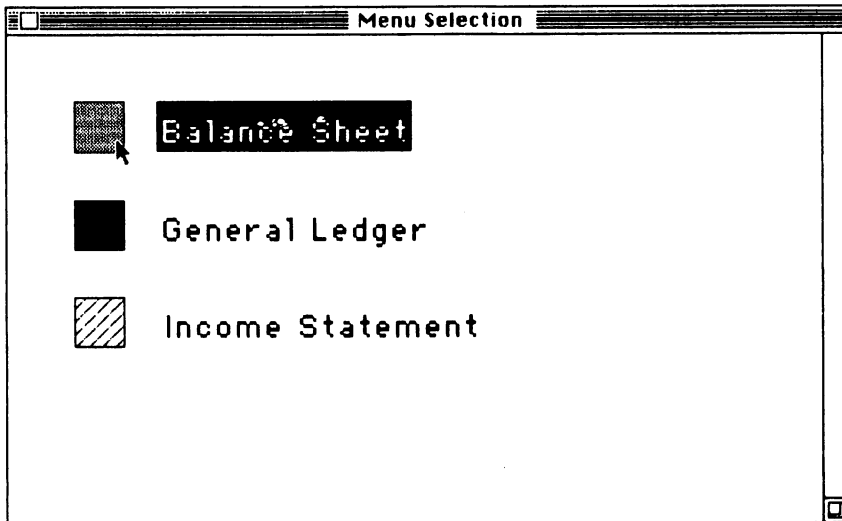


Figure 4-2.
Menu selection

Controlling the Cursor

The mouse interacts with the computer through the *cursor*, an icon tied directly to the mouse hardware. A program has no control over the cursor position; cursor position is entirely a function of the mouse. But your programs can control other aspects of the cursor, such as its shape and how it interacts with figures already on the video display.

DEFINING THE CURSOR

Microsoft BASIC also lets you redesign the shape of the cursor and change how it interacts with the rest of the screen. You can use this capability to remind your program user of the status of the program. In the pen-and-ink example given earlier, for instance, you could change the cursor to a pen shape to indicate when the mouse button is down. Or if a program is erasing things, you could make the cursor look like an eraser. How can you do this?

The most obvious way to start is to define the shape of the cursor. The Mac maintains the shape of the cursor in a grid of 16 rows by 16 columns. The default cursor looks like an arrow. The process of defining your own cursor shape is similar to that of defining a pen pattern, but it is slightly more involved. You still store the patterns in an array. In fact, since the cursor grid is 16 pixels wide and integer variables hold 16 bits, each row of the cursor grid corresponds to one cell of the array where the cursor pattern is stored.

Defining a cursor pattern requires more information than defining a pen pattern. The cursor uses another 16×16 grid of dots as a mask. This mask determines how the cursor changes shape as it is moved over white and black areas of the screen. The defining array must also contain the coordinates of the “hot spot” of the cursor, which identifies the cursor’s pixel position.

Figure 4-3 illustrates the design of a cursor. Notice that the value of each row is calculated the same way it was in Figure 3-20; that is, you add the column values of each blackened cell in a row. For example, the first row of the cursor pattern in Figure 4-3 contains two blackened cells in the columns labeled 256 and 128. Thus, the row value is 256 plus 128, or 384.

Once the data is tucked neatly into the array, your only task is to tell the computer where to find the array with the SETCURSOR call. The following listing illustrates how you can define your own cursor:

```

DEFINT a-z
DIM cur(34)
FOR i=1 TO 34: READ cur(i): NEXT i
DATA 0,384,640,1152,2176,4863,9729,20477,-24579
DATA 20477,9729,4863,2176,1152,640,384
DATA 0,384,640,1152,2176,4863,9729,20477,-24579
DATA 20477,9729,4863,2176,1152,640,384
DATA 10,0
LINE (200,100)-(250,150),,bf
LINE (300,100)-(350,150),,b
SETCURSOR VARPTR(cur(1))
Skip: IF MOUSE(0)=0 THEN Skip

```

When you run this program, the screen cursor changes to the desired shape. Don't worry; you haven't permanently changed the cursor shape. As soon as the program halts, the cursor resumes its familiar shape.

Move the cursor around the screen to get a feel for using the new cursor. When you move over the menu bar, the text shows through the white portion of the cursor. But what if you don't want a transparent cursor?

THE CURSOR MASK

With the cursor mask, you control how the cursor interacts with the current screen. In the preceding program, you used the same shape for the mask as for the cursor. This resulted in the black portion of the cursor staying black, regardless of the screen pixels it passed over, while the white portions were transparent. Table 4-2 contains the rules for changing this setup.

Table 4-2.
Cursor and Mask Effect on the Screen

Cursor Pixel	Mask Pixel	Effect on Screen Pixel
white	black	white
black	black	black
white	white	no change (transparent)
black	white	reverses screen pixel

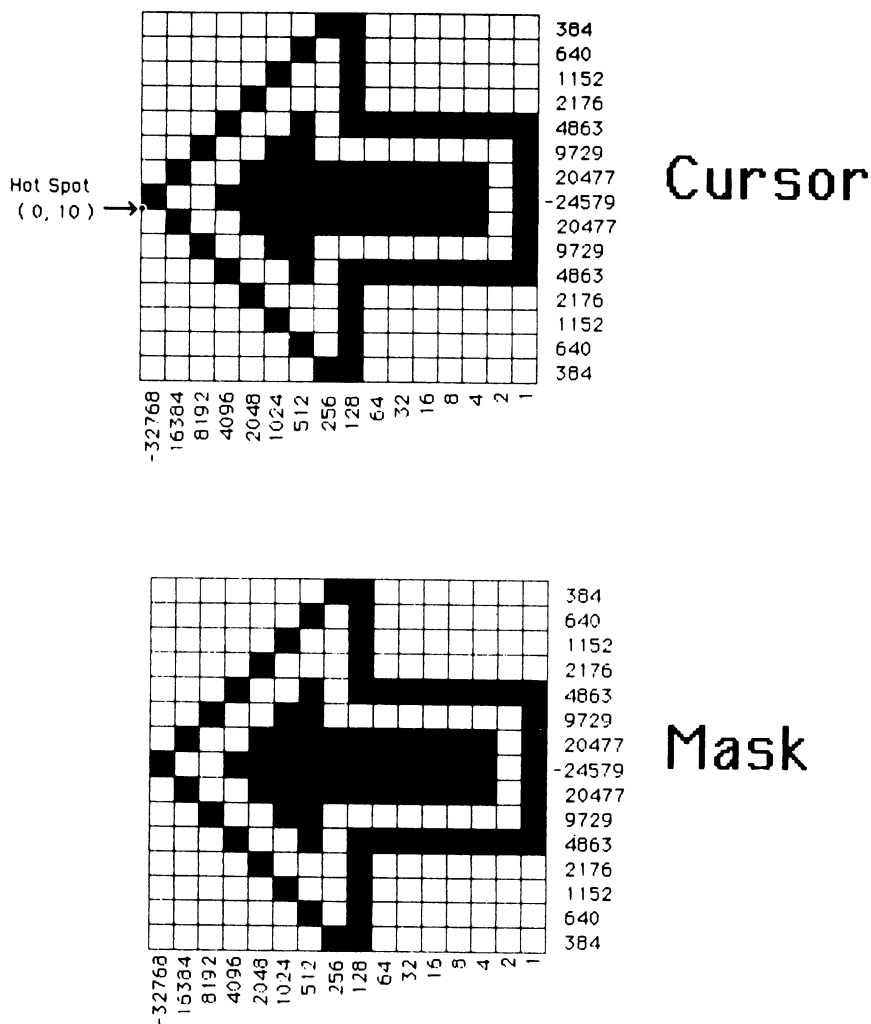


Figure 4-3.
Sample cursor design

Notice in Table 4-2 that if the mask pixel is black, the cursor completely overwrites the screen pixels. If the mask pixel is white, the cursor reacts to the screen. For example, if both the cursor pixel and its corresponding pixel in the mask are black, the screen pixel shows black regardless of its previous color. Of course, these results

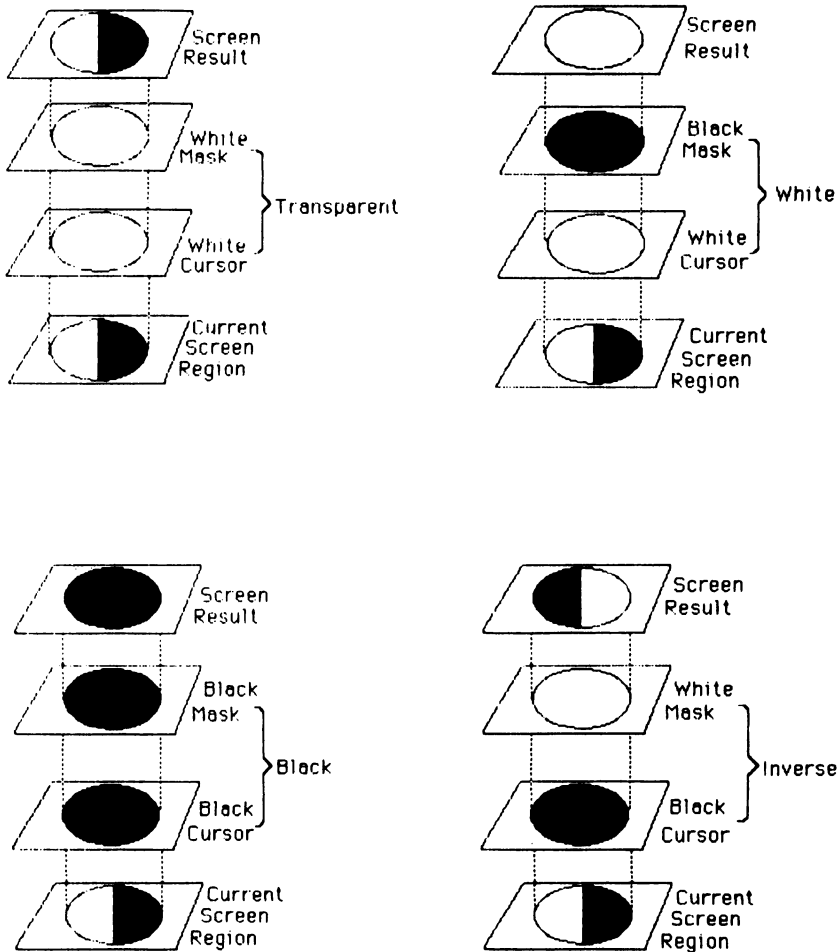


Figure 4-4.
Visualizing cursor effect on screen

are only temporary. As soon as the cursor passes on to another area, the screen pixels revert to their original color. Figure 4-4 illustrates how the mask controls the cursor's effect on the screen.

Let's change the mask so that the interior of the cursor turns white when it is over a black region of the screen.

```

DEFINT a-z
DIM cur(34)
FOR i=1 TO 34: READ cur(i): NEXT i
DATA 0,384,640,1152,2176,4863,9729,20477,-24579
DATA 20477,9729,4863,2176,1152,640,384
DATA 0,384,896,1920,3968,7679,14847,28675,-8189
DATA 28675,14847,7679,3968,1920,896,384
DATA 10,0
LINE (200,100)-(250,150),,bf
LINE (300,100)-(350,150),,b
SETCURSOR VARPTR(cur(1))
Skip: IF MOUSE(0)=0 THEN Skip

```

The next listing shows how the new cursor fits into the menu selection program; the results are shown in Figure 4-5.

```

DEFINT a-z
DIM cur(34)
FOR i=1 TO 34: READ cur(i): NEXT i
DATA 0,384,640,1152,2176,4863,9729,20477,-24579
DATA 20477,9729,4863,2176,1152,640,384
DATA 0,384,896,1920,3968,7679,14847,28675,-8189
DATA 28675,14847,7679,3968,1920,896,384
DATA 10,0
SETCURSOR VARPTR(cur(1))
TEXTFONT 1
TEXTSIZE 18
TEXTFACE 64
DIM pat(12),box(12),edge(12)
FOR i=0 TO 11: READ pat(i): NEXT i
FOR i=0 TO 11: READ box(i): NEXT i
LINE (40,40)-(70,70),,b
LINE (40,100)-(70,130),,b
LINE(40,160)-(70,190),,b
MOVETO 94,65
PRINT "Balance Sheet"
MOVETO 94,125
PRINT "General Ledger"
MOVETO 94,185
PRINT "Income Statement"
DATA 21930,21930,21930,21930,-1,-1,-1,-1
DATA 258,1032,4128,16512
DATA 41,41,70,70,101,41,130,70,161,41,190,70
FILLRECT VARPTR(box(0)),VARPTR(pat(0))
FILLRECT VARPTR(box(4)),VARPTR(pat(4))
FILLRECT VARPTR(box(8)),VARPTR(pat(8))
FOR i=0 TO 11: READ edge(i): NEXT i
DATA 40,90,70,244,100,90,130,255,169,90,190,285
TestClick:
IF MOUSE (0) =0 THEN TestClick
IF MOUSE(3)>40 AND MOUSE(3)<70 AND MOUSE(4)>40 AND

```

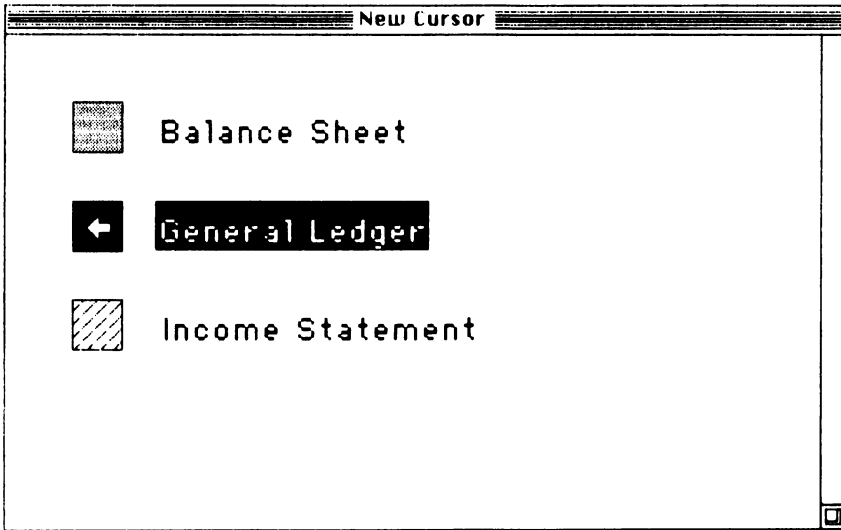


Figure 4-5.
Cursor with revised mask

```

                                MOUSE(4)<70 THEN GOTO Invert1
IF MOUSE(3)>40 AND MOUSE(3)<70 AND MOUSE(4)>100 AND
                                MOUSE(4)<130 THEN GOTO Invert2
IF MOUSE(3)>40 AND MOUSE(3)<70 AND MOUSE(4)>160 AND
                                MOUSE(4)<190 THEN GOTO Invert3

GOTO TestClick
Invert1: INVERTRECT VARPTR(edge(0)): GOTO CleanUp
Invert2: INVERTRECT VARPTR(edge(4)): GOTO CleanUp
Invert3: INVERTRECT VARPTR(edge(8))
CleanUp:
TEXTFONT 3
TEXTSIZE 12
TEXTFACE 0
WHILE MOUSE (0)≠2:WEND

```

MORE CURSOR COMMANDS

There is more to cursor control than changing its shape. For example, if you decide that you want to return to the original cursor, you can use the INITCURSOR routine:

```

DEFINT a-z: DEFSNG x
DIM cur(34)
FOR i=1 TO 34: READ cur(i): NEXT i
DATA 0,384,640,1152,2176,4863,9729,20477,-24579

```

```

DATA 20477,9729,4863,2176,1152,640,384
DATA 0,384,896,1920,3968,7679,14847,28675,-8189
DATA 28675,14847,7679,3968,1920,896,384
DATA 10,0
LINE (200,100)-(250,150),,bf
LINE (300,100)-(350,150),,b
Loop:
INITCURSOR
x=TIMER: WHILE TIMER<=x: WEND
SETCURSOR(VARPTR(cur(1)))
x=TIMER: WHILE TIMER<=x: WEND
IF MOUSE(0)=0 THEN Loop

```

This program flashes two different versions of the cursor with a one-second time delay between them. Press any key to stop the program.

You can also make the cursor temporarily invisible. `HIDECURSOR` turns off the cursor until it is revived by `SHOWCURSOR`. The next listing demonstrates how `HIDECURSOR` works. You can even select menu options while the cursor is invisible.

```

DEFINT a-z: DEFSNG x
DIM cur(34)
FOR i=1 TO 34: READ cur(i): NEXT i
DATA 0,384,640,1152,2176,4863,9729,20477,-24579
DATA 20477,9729,4863,2176,1152,640,384
DATA 0,384,896,1920,3968,7679,14847,28675,-8189
DATA 28675,14847,7679,3968,1920,896,384
DATA 10,0
LINE(200,100)-(250,150),,bf
LINE (300,100)-(350,150),,b
Loop:
INITCURSOR
x=TIMER: WHILE TIMER<=x: WEND
SETCURSOR(VARPTR(cur(1)))
x=TIMER: WHILE TIMER<=x: WEND
HIDECURSOR
x=TIMER: WHILE TIMER<=x: WEND
IF MOUSE(0)=0 THEN Loop

```

Note that although a `SHOWCURSOR` call is not included in the program, the cursor will still be displayed at the start of each loop. This occurs because the `INITCURSOR` call does an automatic `SHOWCURSOR`.

Another way to hide the cursor is with `OBSCURECURSOR`. The cursor stays hidden until the user moves the mouse. You can think of this routine as a `HIDECURSOR` with a mouse-activated `SHOWCURSOR` feature. The following listing shows `OBSCURECURSOR` in action.


```

DEFINT a-z: DEFSNG x
DIM cur(34)
FOR i=1 TO 34: READ cur(i): NEXT i
DATA 0,384,640,1152,2176,4863,9729,20477,-24579
DATA 20477,9729,4863,2176,1152,640,384
DATA 0,384,896,1920,3968,7679,14847,28675,-8189
DATA 28675,14847,7679,3968,1920,896,384
DATA 10,0
LINE(200,100)-(250,150),,bf
LINE(300,100)-(350,150),,b
SETCURSOR(VARPTR(cur(1)))
Loop:
OBSCURECURSOR
x=TIMER: WHILE TIMER<=x: WEND
IF MOUSE(0)=0 THEN Loop

```

POSITIONING THE CURSOR

With all the wonderful things your program can do to control the cursor, there is one thing missing—it cannot control the cursor's position. There is no program statement or ROM call that will let your program position the cursor at a fixed pixel location. All those MOVETO- and LOCATE-type commands affect the *pen's* position, not the cursor's. The only way to move the cursor is with the mouse. The user is in complete control.

Program Interaction

This section deals with other ways of interacting with your BASIC programs. The mouse is terrific for pointing, and the Macintosh has some built-in routines to give you things to point at—namely, menus and dialog boxes with buttons. In addition, Microsoft BASIC supports a clever way to control the process of interaction, called *event trapping*.

EVENT TRAPPING

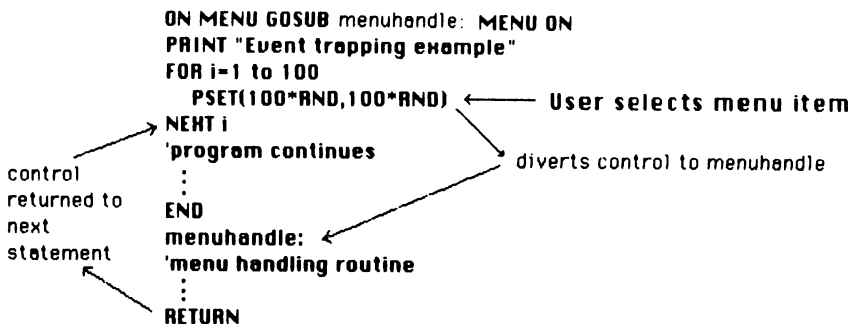
No chapter on interactive graphics for the Mac would be complete without a discussion of event trapping. *Event trapping* consists of detecting certain user actions and responding selectively to them. Events that can be detected on the Mac include mouse activity (MOUSE), menu activity (MENU), dialog activity (DIALOG), timer activity (TIMER), and program interruption (BREAK). For graphics, we will concern ourselves with the first four.

Event trapping lets your program respond instantly to certain user actions. It redirects program flow to other sections of the pro-

gram. For example, when the user clicks in a menu, event trapping can automatically send program control to the subroutine that processes menu selections. Unfortunately, you must still write the subroutines that respond to events.

A word of caution: while event trapping is useful, it lends itself to program bugs and errors, so use it with caution. Observe all the caveats expressed in the BASIC manual. Event trapping also slows execution slightly.

How do you do event trapping? First, turn on the trapping function for a particular event with 'event ON'. For example, MENU ON starts the computer scanning for button activity before it executes each statement. Then use 'ON event GOSUB' to direct program flow to an event-handling subroutine when that event occurs. For example, 'ON MENU GOSUB menuhandle' sends program control to the subroutine menuhandle. The interesting thing about event trapping is that the jump to the subroutine does not occur when these statements are executed. Instead, it happens when event MENU occurs (when you click a menu selection), regardless of the current program line. When the subroutine is finished, control returns to the next program line, as in this illustration:



Other events can be trapped in a similar fashion. Let's look at a few examples.

CREATING MENUS

The first example uses menus to show how event trapping can interrupt the program at any point.

```

MENU 6,0,1,"New Menu"
MENU 6,1,1,"Doodle"
MENU 6,2,1,"Type"
MENU 6,3,1,"Erase"
  
```

```

MENU 6,4,1,"Stop"
ON MENU GOSUB MenuCheck : MENU ON
Loop: GOTO Loop

MenuCheck:
  MENU
  IF MENU(0)<>6 THEN RETURN
  ON MENU(1) GOSUB Doodle, Type, Eraser, Halt
RETURN

Doodle:
  PRINT"doodle routine goes here"
RETURN

Type:
  PRINT"type routine goes here"
RETURN

Eraser:
  PRINT"erase routine goes here"
RETURN

Halt:
  MENU RESET
END

```

The MENU statements in this listing set up a sixth menu, called New Menu, with options Doodle, Type, Erase, and Stop. The ON MENU GOSUB and MENU ON statements are the keys to event trapping. Only one thing happens when they are executed: the computer stores the jump location (menucheck) in memory. No jumps take place. The program settles into an endless loop at statement Loop.

Now the excitement begins. As soon as the user selects a menu item, the computer activates the menu trap. The program breaks out of the endless loop and jumps to the line MenuCheck. The trap is temporarily disabled. When the program gets a RETURN, program control returns to the next program line, and the trap is reactivated; that is, the program resumes its endless loop until the next menu selection is made.

The MenuCheck routine reroutes program control to a section of the program where the menu items are implemented. The MENU statement turns the menu title back to black on white. The IF statement makes sure that the right menu is selected; otherwise, the program returns to Loop. ON/GOSUB sends control to the correct section of the program, based on the menu options 1 through 4.

The current program has dummy PRINT statements in all of the

routines except the halt routine. Try selecting each of the routines; then select Stop. This routine executes a MENU RESET that erases the menu we set up and returns to BASIC's default menu bar. STOP ends the program.

In the last example, the computer trapped menu activity and delivered program control to the appropriate subroutine. The next program fills in the empty subroutines:

```
DIM pen%(2),cur%(34)
GOSUB SetUpCursor
MENU 6,0,1,"New Menu"
MENU 6,1,1,"Doodle"
MENU 6,2,1,"Type"
MENU 6,3,1,"Erase"
MENU 6,4,1,"Stop"
ON MENU GOSUB MenuCheck : MENU ON
Loop: GOTO Loop
```

MenuCheck:

```
ON MENU(1) GOSUB Doodle, Type, Eraser, Halt
IF MENU(0)=0 THEN MenuCheck
RETURN
```

SetUpCursor:

```
FOR i=1 TO 34: READ cur%(i): NEXT i
DATA -1,-32767,-32767,-32767,-32767,-32767,-32767
DATA -32767,-32767,-32767,-32767,-32767,-32767
DATA -32767,-32767,-1,-1,-32767,-32767
DATA -32767,-32767,-32767,-32767,-32767,-32767
DATA -32767,-32767,-32767,-32767,-32767,-32767,-1
DATA 8,8
RETURN
```

Doodle:

```
MENU 6,1,0
DLoop:
MOVETO MOUSE(1),MOUSE(2)
IF MOUSE(0)<0 THEN LINETO MOUSE(1),MOUSE(2)
IF MENU(0)=0 THEN DLoop
MENU 6,1,1
RETURN
```

Type:

```
MENU 6,2,0
TLoop:
IF MOUSE(0)<0 THEN MOVETO MOUSE(3),MOUSE(4)
GETPEN VARPTR(pen%(0)): y=pen%(0): x=pen%(1)
LINE(x,y)-(x,y-10): FOR i=1 TO 100: NEXT i
LINE(x,y)-(x,y-10),30: FOR i=1 TO 100: NEXT i
In$=INKEY$: PRINT In$;
```

```

IF MENU(0)=0 THEN TLoop
MENU 6,2,1
RETURN

```

```

Eraser:
MENU 6,3,0
ELoop:
IF MENU(0)<>0 THEN ExitEraser
IF MOUSE(2)>0 THEN SETCURSOR(VARPTR(cur%{1})) ELSE INITCURSOR
IF MOUSE(0)=0 THEN ELoop:
x=MOUSE(1): y=MOUSE(2)
LINE(x-8,y-8)-(x+7,y+7),30,bf
IF MENU(0)=0 THEN ELoop
ExitEraser:
MENU 6,3,1
RETURN

```

```

Halt:
MENU RESET
END

```

This more complete version rounds out the program. It also demonstrates how you can dim a menu item (as illustrated in Figure 4-6). In the doodle routine, MENU 6,1,0 dims the doodle menu item until another menu item is selected. Then MENU 6,1,1 reactivates this item before returning to the menucheck section.

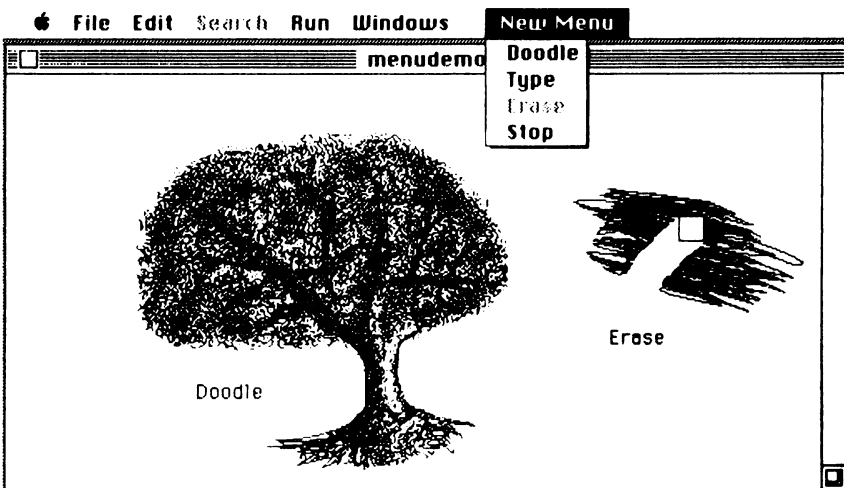


Figure 4-6.
Dimmed menu item

The type text routine uses the INKEY\$ function and PRINT—rather than INPUT—to display text. Although with INPUT, the system handles all the editing features (like positioning the cursor and dragging to select a section of text). INPUT gums up the works in a program where you want to be able to switch quickly from one menu item to another. It maintains program control until you complete the input line by pressing the ENTER or RETURN key. INKEY\$ gives a much faster response; however, the programmer must add any editing features.

The program displays a flashing cursor by alternately drawing black and white vertical lines with the LINE statement. The cursor is located at the point of a mouse click by first setting the pen coordinates to MOUSE(3) and MOUSE(4). As the user types text, the GETPEN function keeps track of the new pen location.

The eraser routine changes the cursor shape whenever the button is pressed and erases with the LINE statement. Again, notice how each menu item is unavailable while that routine is in progress, as shown in Figure 4-6.

USER DIALOG

The Mac also allows the user to interact through the current output window in several ways, all called types of *user dialog*. *Window management* gives the user the ability to create and manipulate different kinds of windows. Special screen areas, called *buttons*, can detect a mouse click (for example, when a user responds to a question). *Edit fields* can handle text input from the user. Finally, a multipartite *DIALOG function* informs the program which dialog events have taken place.

Programs use dialogs to obtain additional information and input from the user. Dialogs are often used like printed forms (such as a tax form or a multiple-choice quiz). For example, when you print a listing, BASIC uses dialogs to inquire about paper size and orientation, print quality, number of copies, and so forth.

Full coverage of dialogs is beyond the scope of this book, but we will discuss some simple implementations. Here is an example that uses several of the above elements:

DrawWindow:

```
WINDOW 2,,(50,50)-(450,280),2
MOVETO 50,50: PRINT "Save drawing as:";
EDIT FIELD 1,"my drawing",(120,70)-(370,85)
MOVETO 80,125: PRINT "Drive:";
BUTTON 1,2,"Internal",(150,112)-(250,125),2
BUTTON 2,1,"External",(150,135)-(250,147),2
```

```

MOVETO 80,172: PRINT "Format:";
BUTTON 3,1,"MacPaint",(150,160)-(250,172),3
BUTTON 4,2,"BASIC",(150,182)-(250,194),3
BUTTON 5,1,"SAVE",(290,130)-(350,150),1
BUTTON 6,1,"CANCEL",(290,165)-(350,185),1

```

GetInput:

```

d=DIALOG(0): IF d=0 THEN GetInput
IF d=6 THEN GOSUB EditName
IF d<>1 THEN GetInput
ON DIALOG(1) GOSUB Internal, External, MacPaint, BASIC, SaveIt, CancelIt
GOTO GetInput
END

```

EditName:

```

title$=EDIT$(1)
EDIT FIELD CLOSE 1
MOVETO 120,85: PRINT title$;
RETURN

```

Internal:

```

BUTTON 1,2
BUTTON 2,1
RETURN

```

External:

```

BUTTON 1,1
BUTTON 2,2
RETURN

```

MacPaint:

```

BUTTON 3,2
BUTTON 4,1
RETURN

```

BASIC:

```

BUTTON 4,2
BUTTON 3,1
RETURN

```

SAVE It:

```

'Save routine goes here
RETURN

```

Cancel It:

```

WINDOW CLOSE 2
END
RETURN

```

DIALOG BOX

The DrawWindow section draws a dialog box for our interaction. This box is window number 2. It resides from (50,50) to (450,280) (absolute screen coordinates). It is a type 2 window (i.e., a dialog box).

Notice that all subsequent screen coordinates use the upper-left corner of this window as location (0,0).

The EDIT FIELD statement creates a screen area used for entering and editing text. This one specifies field number 1, has a default value of 'my drawing', and resides in the rectangle (120,70)-(370,85). Again, these numbers assume that the upper-left corner of the output window is (0,0).

BUTTONS

The rest of the DrawWindow section creates six buttons. A button is so named because it is something you press (click on) to cause a desired action. A Macintosh button contains an area of the screen that is sensitive to a mouse click. The format for the BUTTON statement is BUTTON number, state, title, rectangle, type. The last three parameters are optional. The first BUTTON statement in the listing sets up button number 1 in state 2 (active and selected) with the title 'Internal'; it is sensitive from (150,112) to (250,125) and is type 2 (a check box).

Figure 4-7 shows the three different types of buttons. The first two buttons are check boxes; 1 is selected. The next are called *radio buttons*; 4 is selected. On the right are push buttons; the SAVE button has been selected by the mouse, not by the BUTTON statement.

The GetInput section uses the DIALOG(0) function to detect activ-

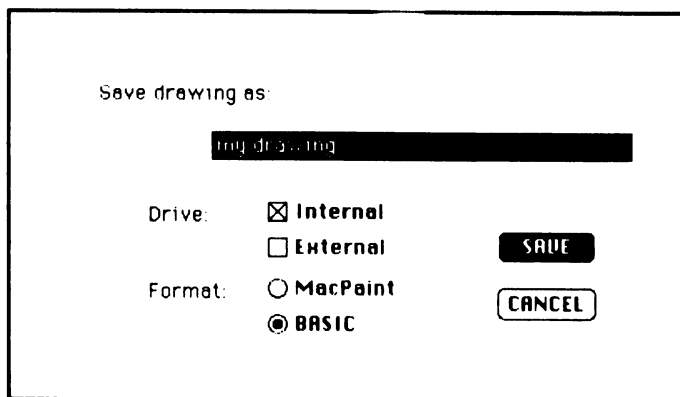


Figure 4-7.
Dialog window

ity by the user. This program uses polling rather than event trapping. Which method you use is a matter of personal preference. The `DIALOG(0)` function returns the values 0 through 7 to indicate the type of activity. The program tests for the values 0 (no dialog activity), 1 (button press), and 6 (RETURN key in edit field). When the user presses the RETURN key, control passes to the `EditName` routine. If a button is pressed, the `ON DIALOG(1)` statement directs control to the appropriate routine. `DIALOG(1)` returns the number of the most recently pressed button.

The edit and button routines are mostly skeletons. The `EditName` routine retrieves the contents of the edit field with `EDIT$` and saves it in `title$`. Then the program closes the edit field. The first four button routines (`Internal`, `External`, `MacPaint`, and `BASIC`) simply change the state of a selected button. Notice that when the user selects one button of a mutually exclusive pair (`Internal` or `External`), the program explicitly turns the other one off. The `CancelIt` routine closes the current output window before stopping the program.

Applications

The following programs are applications of interactive programming.

MAZE

This program draws a maze and then lets the user click the mouse to move along a path, starting from an initial cell.

RANDOMIZE TIMER

DEFINT a-z

`w=30` 'cell width (must be even)

`c=420/w` 'number of columns

`r=280/w` 'number of rows

`s=0`

`DIM m(r+1,c+1), f(2/3*c*r,2)`

DrawMazeGrid:

`FOR x=250-w*c/2 TO 250+w*c/2 STEP w`

`LINE (x,150!-w*r/2)-(x,150!+w*r/2)`

`NEXT x`

`FOR y=150-w*r/2 TO 150+w*r/2 STEP w`

`LINE(250!-w*c/2,y)-(250!+w*c/2,y)`

`NEXT y`

PickInitialCell:

`rz=r*RND(1)+.5: cz=c*RND(1)+.5`

`m(rz,cz)=16: ri=rz: ci=cz`

```

Examine: 'neighbors
rn=rz-1: cn=cz: 'up
IF rn>0 THEN IF m(rn,cn)=0 THEN GOSUB Addtof
rn=rz: cn=cz-1: 'left
IF cn>0 THEN IF m(rn,cn)=0 THEN GOSUB Addtof
rn=rz+1: cn=cz: 'down
IF rn<=r THEN IF m(rn,cn)=0 THEN GOSUB Addtof
rn=rz: cn=cz+1: 'right
IF cn<=c THEN IF m(rn,cn)=0 THEN GOSUB Addtof

'choose a frontier cell
IF s=0 THEN Choose
s1=s*RND(1)*.5: rz=f(s1,1): cz=f(s1,2)
IF rz=0 OR cz=0 THEN Play
f(s1,1)=f(s,1): f(s1,2)=f(s,2): s=s-1

'add cell to maze
ts$="": m(rz,cz)=0
IF m(rz-1,cz)>0 THEN ts$=ts$+"T"
IF m(rz,cz-1)>0 THEN ts$=ts$+"L"
IF m(rz+1,cz)>0 THEN ts$=ts$+"B"
IF m(rz,cz+1)>0 THEN ts$=ts$+"R"
t$=MID$(ts$,LEN(ts$)*RND(1)+.5,1)
WHILE t$="T"
  r1=rz-1: c1=cz
  m(rz,cz)=m(rz,cz)+1: m(r1,c1)=m(r1,c1)+4
  LINE ( 251-w*c/2+w*(c1-1),150-w*r/2+w*r1)-(249-w*c/2+w*c1,
    150-w*r/2+w*r1),30
  t$=""
WEND
WHILE t$="L"
  r1=rz: c1=cz-1
  m(rz,cz)=m(rz,cz)+2: m(r1,c1)=m(r1,c1)+8
  LINE ( 250-w*c/2+w*c1,151-w*r/2+w*(r1-1))-(250-w*c/2+w*c1,
    149-w*r/2+w*r1),30
  t$=""
WEND
WHILE t$="B"
  r1=rz+1: c1=cz
  m(rz,cz)=m(rz,cz)+4: m(r1,c1)=m(r1,c1)+1
  LINE (251-w*c/2+w*(c1-1),150-w*r/2+w*(r1-1))-(249-w*c/2+w*c1,
    150-w*r/2+w*(r1-1)),30
  t$=""
WEND
WHILE t$="R"
  r1=rz: c1=cz+1
  m(rz,cz)=m(rz,cz)+8: m(r1,c1)=m(r1,c1)+2
  LINE ( 250-w*c/2+w*(c1-1),151-w*r/2+w*(r1-1))-(250-w*c/2+w*(c1-1),
    149-w*r/2+w*r1),30
  t$=""

```

WEND

GOTO Examine: ' neighbors

Addtof:

m(m,cn)=-1: s=s+1: f(s,1)=m: f(s,2)=cn

RETURN

Choose: ' start AND finish cells:

m(ri,ci)=m(ri,ci)-16

r1=(r-1)*RND+5

LINE (250-w*c/2,150-w*r/2+w*r1)-(250-w*c/2, 150 -w*r/2+w*(r1-1)),30
temp=r1

r1=(r-1)*RND+5

LINE (250+w*c/2,150-w*r/2+w*r1)-(250+w*c/2, 150 -w*r/2+w*(r1-1)),30

PointStartCell:

PENPAT 2

ulx=(250 - w*c/2) ' upper left corner of the entrance __ x coordinate

uly=(150 - w*r/2 + w*(temp-1)) 'and __ y coordinate

LINE(ulx+3,uly+3)-(ulx+w-3,uly+w-3),,bf

bx=250-w*c/2: by=150-w*r/2

Play:

IF MOUSE(0)=0 **THEN** Play

x=MOUSE(3): y=MOUSE(4)

IF ABS(x-250)>w*c/2 **OR** ABS(y-150)>w*r/2 **THEN** Play

ty=by+INT((y-by)/w)*w

lx=bx+INT((x-bx)/w)*w

flag=0: ft=0: fl=0: fb=0: fr=0

IF POINT(lx+w/2,ty-w/2)=33 **THEN** ft=1

IF POINT(lx-w/2,ty+w/2)=33 **THEN** fl=1

IF POINT(lx+w/2,ty+3*w/2)=33 **THEN** fb=1

IF POINT(lx+3*w/2,ty+w/2)=33 **THEN** fr=1

TestBoundary:

IF ft=1 **AND** POINT(lx+w/2,ty)=30 **THEN** ft=2: flag=1

IF fl=1 **AND** POINT(lx,ty+w/2)=30 **THEN** fl=2: flag=1

IF fb=1 **AND** POINT(lx+w/2,ty+w)=30 **THEN** fb=2: flag=1

IF fr=1 **AND** POINT(lx+w,ty+w/2)=30 **THEN** fr=2: flag=1

PaintNewRect:

IF flag=1 **THEN** **LINE**(lx+3,ty+3)-(lx+w-3,ty+w-3),,bf

IF ft=2 **THEN** **LINE**(lx+3,ty-2)-(lx+w-3,ty+2),,bf

IF fl=2 **THEN** **LINE**(lx-2,ty+3)-(lx+2,ty+w-3),,bf

IF fb=2 **THEN** **LINE**(lx+3,ty+w-2)-(lx+w-3,ty+w+2),,bf

IF fr=2 **THEN** **LINE**(lx+w-2,ty+3)-(lx+w+2,ty+w-3),,bf

IF POINT(lx+w,ty+w/2)=30 **AND** lx+w=250+c*w/2 **AND** flag=1 **THEN** Win

GOTO Play

```

Win:
p(1)=0: p(2)=0: p(3)=342: p(4)=512
FOR i=1 TO 5
  INVERTRECT VARPTR(p(1)): FOR j=1 TO 2000: NEXT j
  INVERTRECT VARPTR(p(1)): FOR j=1 TO 2000: NEXT j
NEXT i
Stay: IF INKEY$="" THEN Stay

```

The maze-drawing portion of the program starts out by drawing a grid and then selects an initial cell at random. Next, the program examines the four neighbors and adds each newly encountered cell to a stack called a *frontier*.

A *stack* is simply a place where you can store data in a "last in, first out" manner. Think of a stack of dishes. Suppose you place a red dish on a stack, then a green dish, then a blue dish. When you take dishes off the stack, you will retrieve them in reverse order: blue, then green, then red.

The program chooses one of the frontier cells and adds it to the maze by removing it from the frontier stack and erasing one of its borders. Control returns to the examine neighbor section, and the program continues until there are no more frontier cells ($s=0$, i.e., the stack is empty). At that point all cells have been added to the maze. Finally, start and end cells are selected at random to complete the maze, and the start cell is highlighted with the `LINE` statement.

The program draws a different maze each time you run it. Notice that you can change the variable `w`, which determines the width and height of the cells; be sure to use an even value.

The interactive part of the program starts at `Play`. The goal is to click your way from the start cell to the end cell. `MOUSE(0)` detects a mouse click; the coordinates are stored in `x` and `y` with `MOUSE(3)` and `MOUSE(4)`. To eliminate points outside of the maze, the program uses the `ABS` function to test the point (x,y) against the midpoint $(250,150)$. The values `ty` and `lx` are set to the top and left boundaries of the selected cell. The program uses these values to test for highlighted neighbors and maze walls.

The program screens each selected cell to make sure that it meets two criteria before adding it to the highlighted path. One, it must have a highlighted neighbor. If so, the appropriate flag `ft`, `fl`, `fb`, or `fr` is set to 1. Two, it must not be separated from this neighbor by a maze wall. Cells that pass both tests are highlighted, as shown in Figure 4-8.

The last `IF` statement in the `PaintNewRect` section tests to see if the maze is complete. If it is, control passes to the `Win` routine, which

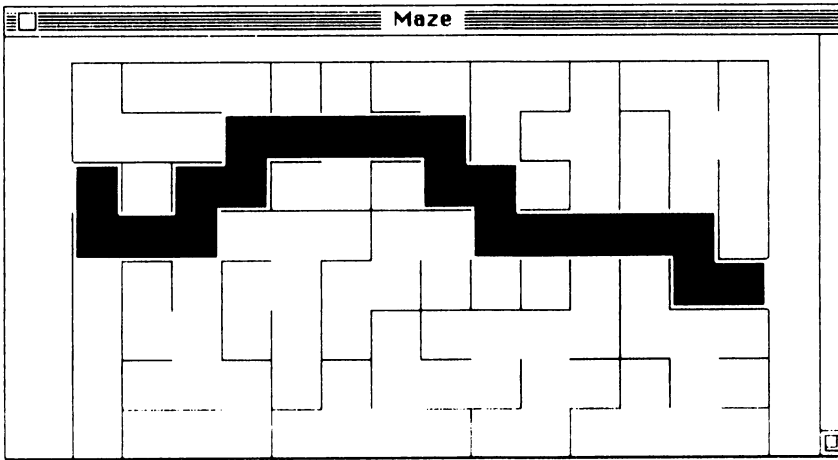


Figure 4-8.
Maze

uses `INVERTRECT` to announce the victory. When the hoopla is over, press any key to stop the program.

PERSONALITY TEST

Here's an application that illustrates the usefulness of mouse selection in testing situations. This example is a personality test that estimates the relative strengths of four personality traits and then graphs them. Don't place too much faith in the results of this program; it's included only to illustrate possibilities:

```
DEFINT a-z: DEFUNG I
DIM rec(32),A$(12),B$(12),C$(12),D$(12),pat(16)
DIM rect(60),det(20),h(20)
FOR i=0 TO 31:READ rec(i):NEXT i
DATA 55,241,70,270,85,241,100,270,115,241,130,270
DATA 145,241,160,270,55,305,70,334,85,305,100,334
DATA 115,305,130,334,145,305,160,334
FOR i=1 TO 12: READ A$(i): NEXT i
DATA self-reliant,bold,aggressive,persistent
DATA determined,brave,competitive,adventurous
DATA decisive,restless,unconquerable,vigorous
FOR i=1 TO 12: READ B$(i): NEXT i
DATA persuasive,open-minded,optimistic,inspiring
DATA confident,convincing,sociable,talkative
DATA playful,attractive,charming,companionable
FOR i=1 TO 12: READ C$(i): NEXT i
DATA loyal,moderate,trusting,obedient,gentle
```

```

DATA controlled,lenient,generous,kind,neighborly
DATA accommodating,good-natured
FOR i=1 TO 12: READ D$(i): NEXT i
DATA humble,accurate,peaceful,adaptable
DATA respectful,cautious,agreeable,precise
DATA god-fearing,soft-spoken,diplomatic,receptive

```

```
NewScreen:
```

```
CLS
```

```
MOVETO 240,40:PRINT "MOST":LINE (237,29)-(274,41),,b
```

```
j=0
```

```
PrintOval:
```

```
FRAMEOVAL VARPTR(rec(j))
```

```
j=j+4
```

```
IF j<13 GOTO PrintOval
```

```
w=w+1
```

```
MOVETO 110,70:PRINT A$(w)
```

```
MOVETO 110,100:PRINT B$(w)
```

```
MOVETO 110,130:PRINT C$(w)
```

```
MOVETO 110,160:PRINT D$(w)
```

```
MOVETO 90,200
```

```
PRINT "Select one word that MOST describes you."
```

```
Loop1: z=MOUSE(0)
```

```
x=MOUSE(3): y=MOUSE(4)
```

```
IF ((x-256)^2/14^2)+((y-62)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(0)):d1=d1+1:
```

```
        GOTO PrintLeast
```

```
IF ((x-256)^2/14^2)+((y-92)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(4)):l1=l1+1:
```

```
        GOTO PrintLeast
```

```
IF ((x-256)^2/14^2)+((y-122)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(8)):s1=s1+1:
```

```
        GOTO PrintLeast
```

```
IF ((x-256)^2/14^2)+((y-152)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(12)):c1=c1+1:
```

```
        GOTO PrintLeast
```

```
GOTO Loop1
```

```
PrintLeast:
```

```
MOVETO 300,40:PRINT "LEAST"
```

```
LINE (297,29)-(340,41),,b: j=16
```

```
PrintOval2:
```

```
FRAMEOVAL VARPTR(rec(j))
```

```
j=j+4:IF j<29 THEN GOTO PrintOval2
```

```
MOVETO 90,200
```

```
PRINT "Select one word that LEAST describes you."
```

```
Loop2:
```

```
z=MOUSE(0)
```

```
x=MOUSE(3): y=MOUSE(4)
```

```
IF ((x-319)^2/14^2)+((y-62)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(16)):d2=d2+1:
```

```
        GOTO Pause
```

```

IF ((x-319)^2/14^2)+((y-92)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(20)):l2=l2+1:
    GOTO Pause
IF ((x-319)^2/14^2)+((y-122)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(24)):s2=s2+1:
    GOTO Pause
IF ((x-319)^2/14^2)+((y-152)^2/7^2)<1 THEN INVERTOVAL VARPTR(rec(28)):c2=c2+1:
    GOTO Pause

GOTO loop2
Pause:
FOR i=1 TO 5000: NEXT i
IF w<12 THEN NewScreen
FOR i=0 TO 15
    READ pot(i)
NEXT i
DATA -32446,9240,6180,17025,27647,-18945,-8581,-8329
DATA -32512,24,6144,129,-7262,-7396,5148,-7262
det(1)=d1-d2
det(2)=l1-l2
det(3)=s1-s2
det(4)=c1-c2
dmax=det(1): dmin=det(1)
CLS
n=4
FOR i=1 TO n
    IF det(i)<dmin THEN dmin=det(i)
    IF det(i)>dmax THEN dmax=det(i)
NEXT i
IF dmax<0 THEN dmax=0
IF dmin>0 THEN dmin=0
l=dmax
FOR i=50 TO 250 STEP 20
    MOVETO 10,i+4
    PRINT USING"####.##";l,
    l=l-(dmax-dmin)/10
    LINE(58,i)-(60,i)
NEXT i
LINE(60,50)-(60,250)
baseline=dmax/(dmax-dmin)*200+50
IF dmin>0 THEN baseline=250
FOR i=1 TO n
    h(i)=baseline-det(i)/(dmax-dmin)*200
NEXT i
LINE(60,baseline)-(470,baseline)
FOR i=1 TO n
    LINE(70+(400/n)*(i-1),h(i))-(70+(400/n)*i-10,baseline),,b rect(4*(i-1))=h(i)+1
    IF det(i)<0 THEN rect(4*(i-1))=baseline+1
    rect(4*(i-1)+1)=70+(400/n)*(i-1)+1

```









```

rect(4*(i-1)*2)=baseline
IF dot(i)<0 THEN rect(4*(i-1)*2)=h(i)
rect(4*(i-1)*3)=70+(400/n)*i-10
NEXT i
MOVETO 81,15:PRINT "DOMINANCE"
MOVETO 179,15:PRINT "INFLUENCE"
MOVETO 281,15:PRINT "STEADINESS"
MOVETO 384,15:PRINT "COMPLIANCE"
FOR r=0 TO n*4 STEP 4
  FILLRECT VARPTR(rect(r)),VARPTR(pat(r))
NEXT r
Stay: IF INKEY$="" THEN Stay

```

Although it might be easier to use radio buttons for word selection, this program uses elliptical buttons. Those of you with a mathematical bent can use other conic sections (circles, parabolas, and hyperbolas) to simplify testing for regions.

The program displays a list of words, four at a time, as shown in Figure 4-9. For each word group, the user selects the two words that best describe and least describe himself or herself. When the entire list has been processed, the program plots the difference between the total of most and least responses for each trait (Figure 4-10). The chart is plotted with a modified version of the bar chart program from Chapter 3.

	MOST	LEAST
persistent		
inspiring		
obedient		
adaptable		

Select one word that LEAST describes you.

Figure 4-9.
Word selection

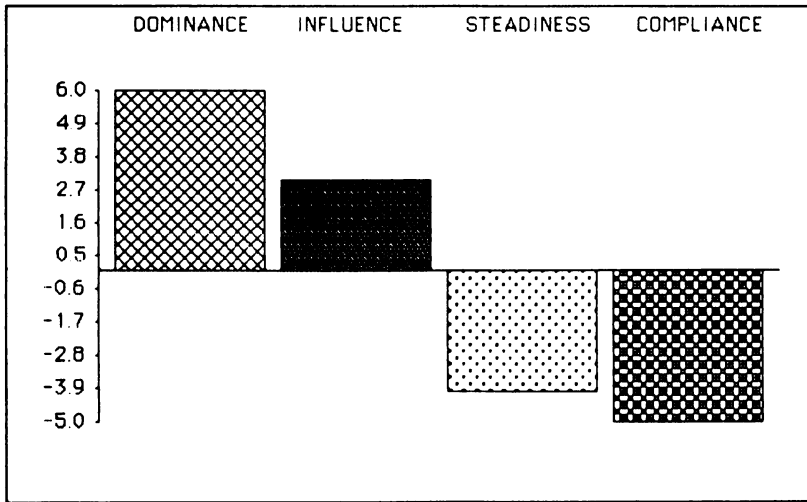


Figure 4-10.
Plot of personality traits

Again, use this program for fun, but please don't make any serious decisions based on the test results.

VISUALLY DIRECTED INSTRUCTION

This program illustrates a very visual way of directing the flow of an educational program. When the student clicks on the desired portion of a picture, the program responds by giving more detailed information. This concept can be extended to several levels of detail, with different options at each level. Here is the program:

```

DEFINT a-z
DIM m(23),s(4),k(25),o(33)

'read mouse data
FOR i=0 TO 32
  READ o(i)
NEXT i
DATA 66,106,100,115,110,115,100,112,100,106,106,106
DATA 116,109,116,115,110,115,100,112,100,112,110,106
DATA 116,106,113,109,111,109,104,106,108

```

'read the keyboard data

FOR i=0 TO 24

 READ k(i)

NEXT i

DATA 50,127,40,140,90,140,40,137,40,127,50,127,90,130

DATA 90,140,80,140,40,137,40,137,80,127,90

'read screen data

FOR i=0 TO 3

 READ s(i)

NEXT i

DATA 65,63,85,87

'read mac body data

FOR i=0 TO 22

 READ m(i)

NEXT i

DATA 46,40,50,80,80,60,60,100,60,100,90,60,90,60,60,50

DATA 70,50,100,90,100,100,90

Draw:

'draw the mac body

CLS

FRAMEPOLY VARPTR(m(0))

FRAMEROUNRECT VARPTR(s(0)),10,10

LINE (60,97)-(90,97)

LINE (80,92)-(87,92)

'draw the keyboard

FRAMEPOLY VARPTR(k(0))

FOR i=44 TO 78 STEP 4

 LINE(i,135)-(i+5,129)

NEXT i

i=52 : j= 129

FOR l=1 TO 4

 LINE(i,j)-(i+28,j) : i= i - 2 : j= j + 2

NEXT l

LINE(54,127)-(88,99)

LINE(52,127)-(86,99)

'draw the mouse

FRAMEPOLY VARPTR(o(0))

LINE(109,105)-(113,105)

LINE(111,104)-(100,84)

MOVETO 140,150

PRINT "CLICK ON ANY COMPONENT ."

Start:

IF (MOUSE(0)>0) THEN GOSUB Check

GOTO Start

Finish:

```
CLS
END
```

Check:

```
IF (MOUSE(1)>60 AND MOUSE(1)<90) AND (MOUSE(2)>60 AND MOUSE(2)<90) THEN
    GOSUB Mac
IF (MOUSE(1)>100 AND MOUSE(1)<110) AND (MOUSE(2)>106 AND MOUSE(2)<115) THEN
    GOSUB Mouse1
IF (MOUSE(1)>40 AND MOUSE(1)<90) AND (MOUSE(2)>127 AND MOUSE(2)<140) THEN
    GOSUB Keyboard
RETURN
```

Mac:

```
CLS
PRINT "THE MACINTOSH SCREEN "
PRINT "The Macintosh screen has a resolution of"
PRINT "512x342. All the information needed to "
PRINT "generate the screen image is stored in a"
PRINT "special part of random access memory called"
PRINT "video RAM. The screen is bit-mapped. In other"
PRINT "words, each pixel corresponds to a bit in the"
PRINT "video RAM. Drawing bit mapped pictures"
PRINT "requires a lot of computing power."
PRINT
INPUT "DO YOU WANT TO GET MORE INFO ? (Y/N)" ,a$
IF UCASE$(a$) = "Y" THEN GOTO Draw ELSE GOTO Finish
RETURN
```

MOUSE1:

```
CLS
PRINT "THE MOUSE"
PRINT "The Mac uses is a mechanical/optical mouse."
PRINT "The roller inside is mechanically coupled"
PRINT "to two rotating vanes that interrupt beams"
PRINT "from light-emitting diodes that light up"
PRINT "phototransistors. The two vanes track"
PRINT "vertical and horizontal motions. The mouse"
PRINT "can sense only absolute location. That is,"
PRINT "the way you hold the mouse is not important"
PRINT "as far as the direction is concerned."
PRINT
INPUT "DO YOU WANT TO GET MORE INFO ? (Y/N)" ,a$
IF UCASE$(a$) = "Y" THEN GOTO Draw ELSE GOTO Finish
RETURN
```

Keyboard:

```
CLS
```

```

PRINT "THE KEYBOARD"
PRINT "The Macintosh keyboard has the standard QWERTY"
PRINT "layout. It has some special keys such as Command"
PRINT "and Option. You can get foreign characters and"
PRINT "graphics symbols by pressing the OPTION key."
PRINT "You can adjust the features of your keyboard by"
PRINT "choosing the control panel from the apple menu."
PRINT
INPUT "DO YOU WANT TO GET MORE INFO ? (Y/N)" ,a$
IF UCASE$(a$) = "Y" THEN GOTO Draw ELSE GOTO Finish
RETURN

```

In the simplified example this program presents, there are three components of a computer system (Figure 4-11). The student clicks on a component to obtain more information about it. The components are drawn with the BASIC LINE statement and the FRAMEPOLY ROM call.

In Chapter 6 you will learn how to use MacPaint figures in your BASIC programs; they will greatly reduce the effort required to implement this kind of program.

This program uses rectangular screen regions to test for selection. They only roughly approximate the screen figures. You could use more precise test shapes if desired.

CURSOR EDITOR

Defining patterns and cursors with pencil and paper gets tedious quickly. That's why you typed a pattern utility program in the last chapter. The following cursor editor program is based on the same idea, but it has a few twists.

```

INITCURSOR
DEFINT A-Z
DIM Cursor(33),GridMark(15,31),UndoBuff(15,31)
Alive = -1 : Dead = 0 : E = 14 : F = 15 : Sx = 16
Page = 256 : MaxINT = 32767 : Real# = 65536!
GOSUB CreateMenus
WINDOW 1,,(120,35)-(375,290),4
ON MENU GOSUB HandleMenu: MENU ON

WaitForFinger:
IF MOUSE(0)<>0 THEN WaitForFinger
WaitForMouse:
IF MOUSE(0)=Dead THEN WaitForMouse
X = MOUSE(3)\Sx: Y = MOUSE(4)\Sx
IF (X<0 OR X>F) OR (Y<0 OR Y>F) THEN WaitForFinger
ON Mode GOSUB MakeMark,MakeMask
GOTO WaitForFinger

```

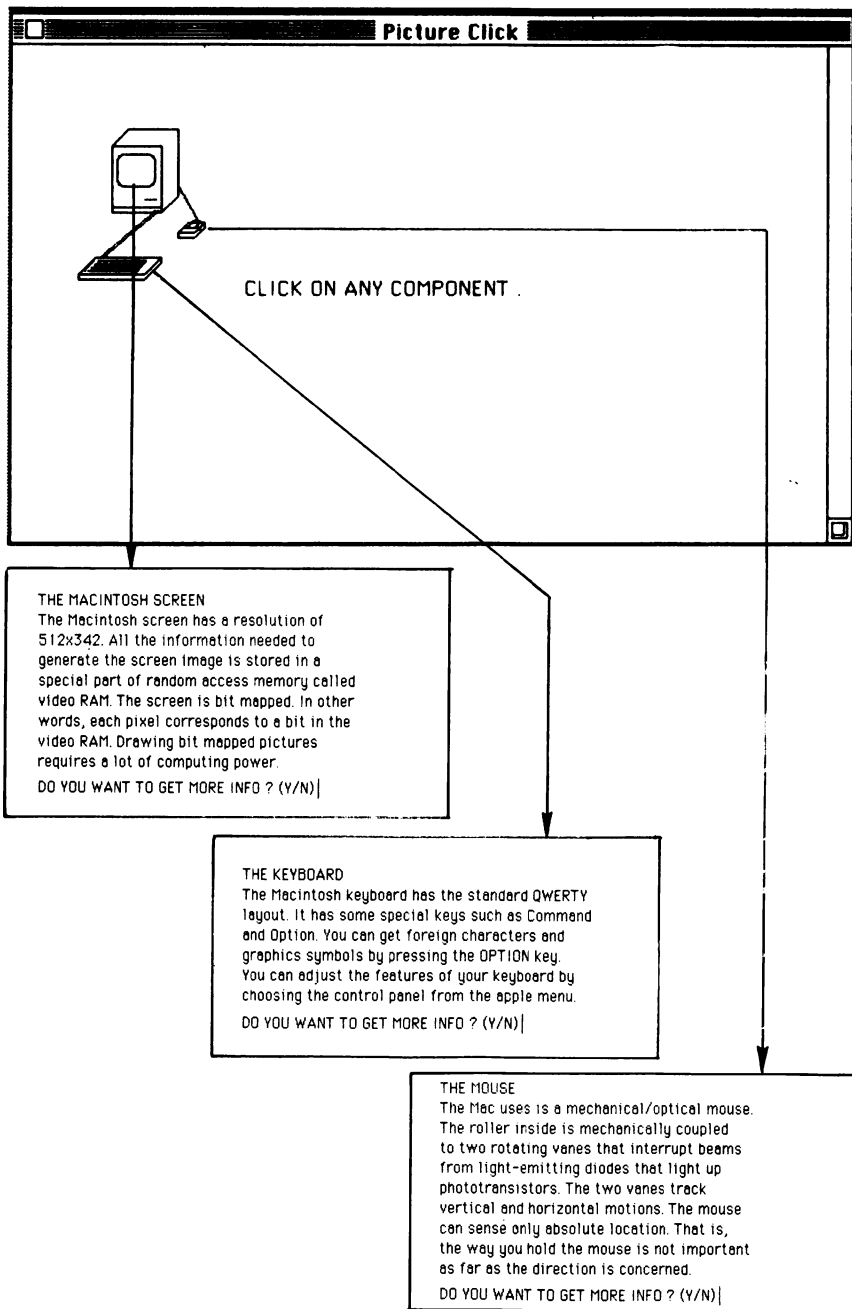


Figure 4-11.
 Obtaining additional information by clicking on the screen

MakeMark:

```
GridMark(X,Y) = NOT GridMark(X,Y)
GOTO MarkXY
```

MakeMask:

```
GridMark(X,Y+Sx) = NOT GridMark(X,Y+Sx)
GOTO MaskXY
```

HandleMenu:

```
Pulled = MENU(0) : Item = MENU(1)
ON Pulled GOSUB File,EditMenu,ModeChange,Help,Quit
MENU
IF NOT ReDraw THEN RETURN
RestoreGrid:
PENMODE 8
CLS
ReDraw = 0
FOR X = F TO Page-F STEP Sx
  LINE(X,0)-(X,Page)
  LINE(0,X)-(Page,X)
NEXT
PENMODE 10
FOR X = 0 TO F
  FOR Y = 0 TO F
    IF GridMark(X,Y) THEN GOSUB MarkXY
    IF GridMark(X,Y+Sx) THEN GOSUB MaskXY
  NEXT
NEXT
RETURN
```

MarkXY:

```
Box%(0)= Y*Sx: Box%(1)=X*Sx
Box%(2)=Y*Sx+F: Box%(3)=X*Sx+F
INVERTRECT (VARPTR(Box%(0)))
RETURN
```

MaskXY:

```
Box%(0)= Y*Sx+2: Box%(1)=X*Sx+2
Box%(2)=Y*Sx+13: Box%(3)=X*Sx+13
DoOval:
INVERTOVAL (VARPTR(Box%(0)))
RETURN
```

File:

```
IF Item = 2 THEN SaveCursor
CFile$ = FILES$(1,"CRSR")
ReDraw = -1
IF CFile$="" THEN RETURN ELSE CLS
OPEN "I", #1,CFile$
FOR I = 0 TO 33
  INPUT #1, Cursor(I)
```

```

NEXT
CLOSE
FOR Y = 0 TO F : DBits# = 0 : MBits = 0
  FOR X = F TO 0 STEP -1
    Bits# = 2^X
    IF Bits# > MaxINT THEN Bits# = Bits# - Real#
    GridMark(F-X,Y) = ((Cursor(Y) AND Bits#) = Bits#)
    GridMark(F-X,Y+Sx) = ((Cursor(Y+Sx) AND Bits#) = Bits#)
  NEXT
NEXT
HotX = Cursor(33): HotY = Cursor(32)
RETURN

```

```

SaveCursor:
CFile$ = FILES$(0,"Save cursor data as:")
ReDraw = -1
IF CFile$="" THEN RETURN
CLS
GOSUB Update
OPEN "O", #1,CFile$
FOR I = 0 TO 33
  PRINT #1,Cursor(I)
NEXT
CLOSE
NAME CFile$ AS CFile$,"CRSR"
RETURN

```

```

InitGrid:
Mode = 1
FOR X = 0 TO F
  FOR Y = 0 TO 31
    GridMark(X,Y) = 0
    UndoBuff(X,Y) = 0
  NEXT
NEXT
Item = Mode
GOSUB ModeChange
GOTO RestoreGrid

```

```

CreateMenus:
MENU 1,0,1,"File"
MENU 1,1,1,"Get Cursor"
MENU 1,2,1,"Save Cursor"

MENU 2,0,1,"Edit"
MENU 2,1,1,"Set Cursor"
MENU 2,2,1,"Normal Cursor"
MENU 2,3,1,"Change Hot Spot"
MENU 2,4,1,"Undo"

```

MENU 2,5,1,"Erase Pattern"

MENU 4,0,1,"Help"

MENU 4,1,1,"Instructions"

MENU 5,0,1,"Quit"

MENU 5,1,1,"Let Me Outta here!"

ModeChange:

Mode = Item

MENU 3,0,1,"Mode"

MENU 3,1,1+ABS(Mode=1),"Edit Cursor"

MENU 3,2,1+ABS(Mode=2),"Edit Mask"

RETURN

EditMenu:

ON Item GOTO ChangeIt,Pointer,GetHotXY,Undo,InitGrid

Undo:

FOR X = 0 TO F

FOR Y = 0 TO 31

SWAP GridMark(X,Y),UndoBuff(X,Y)

NEXT

NEXT

GOTO RestoreGrid

Update:

FOR Y = 0 TO F

DBits# = 0 : MBits# = 0

FOR X = F TO 0 STEP -1

UndoBuff(X,Y)=GridMark(X,Y)

UndoBuff(X,Y+Sx)=GridMark(X,Y+Sx)

IF GridMark(F-X,Y) THEN DBits#=DBits#+2*X

IF GridMark(F-X,Y+Sx) THEN MBits#=MBits#+2*X

NEXT

IF DBits# > MaxINT THEN DBits# = DBits#-Real#

IF MBits# > MaxINT THEN MBits# = MBits#-Real#

Cursor(Y)=DBits# : Cursor(Y+Sx)=MBits#

NEXT

Cursor(33) = HotX : Cursor(32) =HotY

RETURN

ChangeIt:

GOSUB Update

SETCURSOR (VARPTR(Cursor(0)))

RETURN

Pointer:

INITCURSOR

RETURN

GetHotXY:

WINDOW 2,,(8,25)-(100,100),4


```

EDIT FIELD 1,STR$(HotX),(5,10)-(30,25),,2
EDIT FIELD 2,STR$(HotY),(5,30)-(30,45),,2
BUTTON 1,1,"Ok",(20,55)-(80,70)
EDIT FIELD 1: TEXTFONT 0
MOVETO 35,23: PRINT "X (Horiz)"
MOVETO 35,43: PRINT "Y (Vert)"

Loop:
Action = DIALOG(0)
IF Action = 1 THEN EndLoop
IF Action = 2 THEN 1 = DIALOG(2): EDIT FIELD 1
GOTO Loop
EndLoop:
HotX = VAL(EDIT$(1))
  HotY = VAL (EDIT $(2))
  WINDOW CLOSE 2
  WINDOW OUTPUT 1
RETURN

Help:
RETURN

Quit:
MENU RESET
MENU OFF
END

```

This program is largely controlled by menus. It uses the MOUSE functions, event trapping, and even such dialog features as windows and edit fields. In addition, the program allows you to save your cursors as files on disk and to load them back in for editing. You should be able to use the File routine in this program as a guideline for loading cursor files into your own programs.

Figure 4-12 illustrates a sample cursor. The black squares represent cursor pixels and the black circles represent mask pixels.

TOWERS OF HANOI

Towers of Hanoi is a puzzle used as an example of recursive programming technique in nearly every college-level computer course in this country. The object of the puzzle is to move all of the disks on the leftmost tower to the right tower without placing a large disk on a smaller one.

```

DEFINT A-Z
True = -1 : False = 0 : Count = 4
DIM Disc(2,Count-1): DIM Level(Count+5)
GOSUB Setup
GOSUB InitDiscs

```

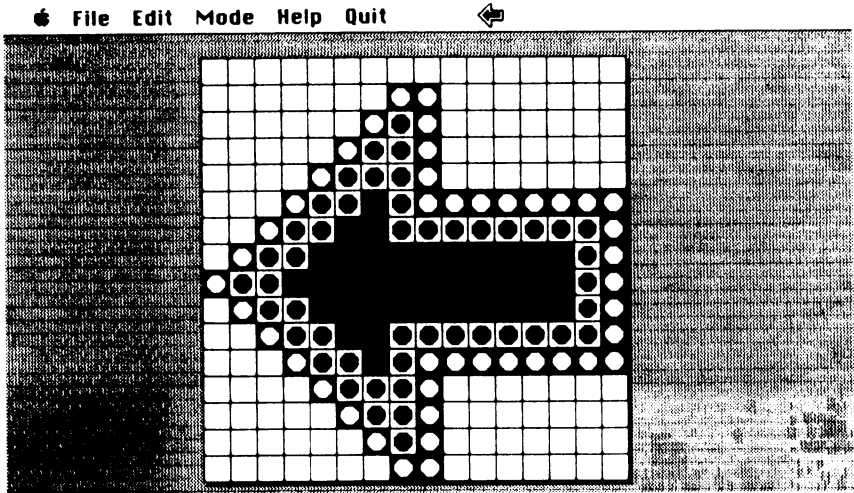


Figure 4-12.
Cursor editor

ContinuePlay:

```

GOSUB GetDisc
GOSUB FindTower
GOSUB MoveDisc
Moves = Moves + 1
MOVETO 10,10 : PRINT Moves
GOTO ContinuePlay

```

MoveDisc:

```

X = NewTower: Y = NewLevel
XX = OldTower: YY = OldLevel
SWAP Disc(XX,YY),Disc(X,Y)
GOSUB PlaceDisc
LINE (78+XX*110,155+YY*19)-(88+XX*110,173+YY*19),,BF
Tower(DiskToMove)=NewTower: Level(DiskToMove)= NewLevel
RETURN

```

AbortDisc:

```

GOSUB GetDisc
FindTower:
Msg$ = "Select A Tower"
GOSUB DoMsg
Okay = False
OldTower = Tower(DiskToMove)
OldLevel = Level(DiskToMove)

```

```

WHILE NOT Okay
  Chosen = 4
  WHILE Chosen > 3
    GOSUB GetButton
    IF Chosen = 4 THEN Restart
    IF Chosen = 5 THEN Quit
  WEND
  Picked = Chosen-1
  IF Picked = OldTower THEN AbortDisc
  IF Disc(Picked,Count-1)=0 THEN NewLevel=Count-1: GOTO Valid
  FOR Y=Count-1 TO 1 STEP -1
    IF Disc(Picked,Y-1)=0 AND Disc(Picked,Y) > DiskToMove THEN NewLevel=Y-1:
      Okay=True: Y=0
  NEXT
  IgnoreTest:
WEND
Valid:
  NewTower = Picked
RETURN

```

```

GetDisc:
  Msg$ = " Select A Disc"
  GOSUB DoMsg
  Okay = False
  WHILE NOT Okay
    Chosen = 0
    WHILE Chosen < 4
      GOSUB GetButton
      IF Chosen = 4 THEN Restart
      IF Chosen = 5 THEN Quit
    WEND
    DiskToMove = Chosen - 5
    IF Level(DiskToMove) = 0 THEN Leave
    IF Disc(Tower(DiskToMove),Level(DiskToMove)-1)=0 THEN Okay=True
  WEND
  Leave:
RETURN

```

```

GetButton:
  WHILE DIALOG(0) <> 1: WEND
  Chosen = DIALOG(1)
RETURN

```

```

DoMsg:
  LINE (91,32)-(299,56),30,BF
  MOVETO 140,49: PRINT Msg$
RETURN

```

PutDiscs:

```

ID = 0
FOR I = 78 TO 298 STEP 110
  ID = ID + 1
  LINE (I,80)-(I+10,230),,BF
  BUTTON ID,1,"",(I-3,67)-(I+15,81),3
NEXT
FOR X = 0 TO 2
  FOR Y = 0 TO Count-1
    IF Disc(X,Y)<>0 THEN GOSUB PlaceDisc
  DoNextDisc:
  NEXT Y
NEXT X
RETURN

```

PlaceDisc:

```

ID = Disc(X,Y) : Size = (40 + ID * 10)/2
TopX = 84 + X * 110 - Size: TopY = 155 + Y * 19
BotX = 84 + X * 110 + Size: BotY = 174 + Y * 19
BUTTON ID+5,1,STR$(ID),(TopX,TopY)-(BotX,BotY)
RETURN

```

InitDiscs:

```

FOR X = 1 TO 2: FOR Y = 0 TO Count-1
  Disc(X,Y)=0
NEXT Y,X
FOR Y = 1 TO Count
  Disc(0,Y-1)=Y
  Tower(Y)=0
  Level(Y)=Y-1
NEXT
GOTO PutDiscs

```

Setup:

```

WINDOW 1,,(60,40)-(452,320),2
MOVETO 90,20 : TEXTFONT 0 : TEXTSIZE 24
PRINT "Towers Of Hanoi": TEXTSIZE 12
LINE (90,31)-(300,57),,B
BUTTON 4,1,"Restart",(5,255)-(80,271)
BUTTON 5,1,"Quit",(311,255)-(386,271)
RETURN

```

Restart:

```

RUN

```

Quit:

```

WINDOW CLOSE 1
END

```

This implementation represents the disks with push buttons and uses radio buttons for tower selection, as shown in Figure 4-13.

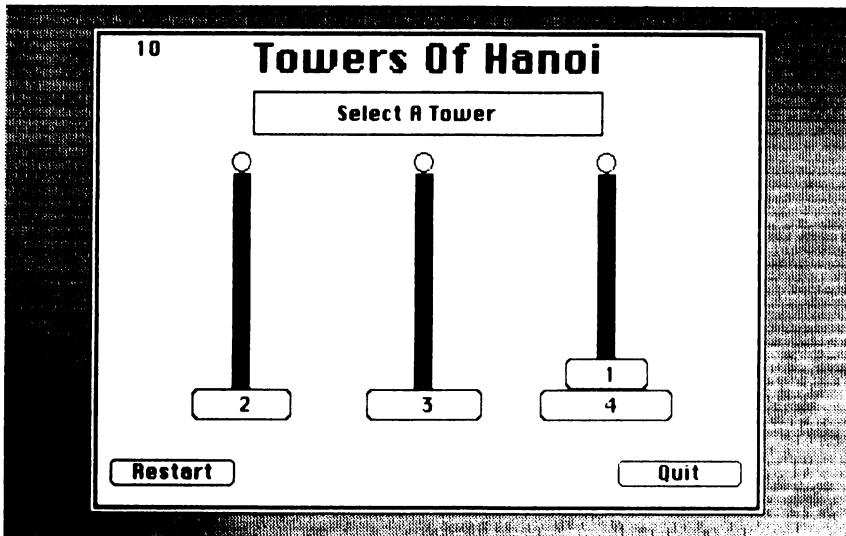


Figure 4-13.
Towers of Hanoi

Summary

This chapter has introduced four major types of interactive graphics: mouse input, controlling the cursor, program interaction, and applications. These are powerful tools for making your graphics programs responsive to user input.

The next chapter investigates how to add sound to your programs.

BASIC Statements

ABS
 BUTTON
 CLOSE
 DIALOG
 EDIT FIELD
 EDIT\$
 MENU
 MENU OFF
 MENU RESET
 MOUSE
 NAME

BASIC Statements, *Continued*

NOT
ON/GOSUB
ON/GOTO
OPEN
PRINT #
SWAP
UCASE\$
WEND
WHILE
WINDOW
WINDOW CLOSE
WINDOW OUTPUT

ROM Calls

GETPEN
HIDECURSOR
INITCURSOR
PENMODE
PENNORMAL
PENPAT
SETCURSOR
OBSCURECURSOR
SHOWCURSOR

5

Sound

Using sound in games and graphics presentations adds excitement to your programs. The Macintosh has one of the most sophisticated sound synthesizers available in microcomputers today. This chapter discusses the sounds you can create using Microsoft BASIC and your Macintosh. It also shows you how you can integrate them into your graphics programs to create your own micro-orchestra.

A knowledge of music theory is not necessary to exploit the Mac's audio potential. This chapter provides a number of sample programs to base your experimentation on. Be sure to experiment, and most of all, have fun!

To take full advantage of the sound features of your Macintosh, you'll need a little background information on sound in general and the Mac synthesizer in particular. The following sections provide you with all you need to know.

The Nature of Sound

Audio systems use speakers to produce sound. The speakers convert electrical impulses into vibrations that move the air around the speaker. These vibrations, or waves, travel through the air to your ear, causing your eardrums to vibrate. Your brain then interprets these vibrations as voices, music, or noise.

The complex structure of all sounds (even those of musical instruments) can be broken into its components in the form of *sine waves*, as shown in Figure 5-1. The shape of the sine wave is the same as its mathematical function. (You may want to dust off your high school trigonometry book if you need a refresher.) A complex sound structure can be synthesized by adding a number of sine waves of different amplitudes, wavelengths (or frequencies), and phases.

Let's look at these key terms. Notice that the wave moves above and below a center line. The wave's *amplitude* is the distance from the highest (or lowest) point to the center line. The amplitude determines the sound's intensity: the greater the amplitude, the louder the sound. The *wavelength* is the distance between the wave's peaks (or valleys, in this illustration). One round trip from peak to peak (or

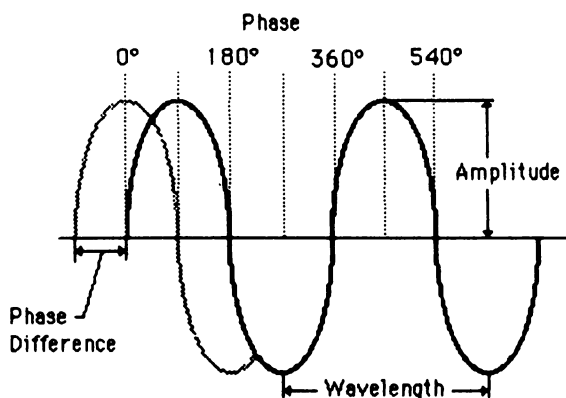


Figure 5-1.
Sine waves

valley to valley) is referred to as a *cycle*. The *phase* of a wave describes its position in time or in relation to another wave. A wave cycle is divided into 360 degrees of phase. Figure 5-1 has a second wave which is 90 degrees out of phase with the main wave. This difference is referred to as the *phase difference* or *phase shift*.

A wave's *frequency* is the number of its cycles that can occur in a given period of time. Frequency is measured in cycles per second or *Hertz (Hz)*. Frequency varies inversely with the wavelength; that is, as the frequency increases, the wavelength decreases. Audible sound ranges from frequencies of 20 Hz to 20,000 Hz. A 20,000-Hz tone has a wavelength of only 0.65 inches, while a 20-Hz tone is 54 feet in wavelength. Frequencies are usually expressed in terms of *pitch*; thus, a higher-pitched tone has a greater frequency than a lower-pitched one.

Music Terminology

To better understand what you will be learning in this chapter, a brief discussion of basic music terminology is also in order. A tone, or *note*, represents a specific frequency of sound. In musical notation, each note is represented by a letter from A to G. The letter indicates where the note lies within an octave. An *octave* is a sequence of eight notes. It starts at any given note and ends at the next occurrence of that same note, eight notes away. The frequency of a note in the higher octave is exactly twice that of the same note in the lower octave. Table 5-1 lists the notes and their frequencies. The standardized relationship of notes and frequencies is known as Ptolemy's diatonic scale.

You'll notice in the table that there are notes labeled D^b, E^b, G^b, A^b, and B^b. The symbol "b" after the note indicates it is a flat. A *flat* note is a note that falls between two "natural" notes. For example, E^b falls between D and E. The same note (E^b) can also be labeled D#. The symbol "#" stands for *sharp*. Like a flat, a sharp note falls between two natural notes. The difference between a sharp and a flat is that the sharp occurs above the note's natural frequency, while a flat occurs below a note's natural frequency. Thus, the note D^b lies below D, and between C and D. The note D# lies above D, and between D and E. Whether you call a note D# or E^b is really up to you; they are both the same note.

The speed at which notes are played is called *tempo*. The note

Table 5-1.
Notes and Their Frequencies

Octave	Note	Frequency (Hz)
3 below middle C	C	33
3 below middle C	D ^b	35.2
3 below middle C	D	37.125
3 below middle C	E ^b	39.6
3 below middle C	E	41.25
3 below middle C	F	44
3 below middle C	G ^b	46.9375
3 below middle C	G	49.5
3 below middle C	A ^b	52.8
3 below middle C	A	55
3 below middle C	B ^b	57.75
3 below middle C	B	61.875
2 below middle C	C	66
2 below middle C	D ^b	70.4
2 below middle C	D	74.25
2 below middle C	E ^b	79.2
2 below middle C	E	82.5
2 below middle C	F	88
2 below middle C	G ^b	93.875
2 below middle C	G	99
2 below middle C	A ^b	105.6
2 below middle C	A	110
2 below middle C	B ^b	115.5
2 below middle C	B	123.75
1 below middle C	C	132
1 below middle C	D ^b	140.8
1 below middle C	D	148.5
1 below middle C	E ^b	158.4
1 below middle C	E	165
1 below middle C	F	176
1 below middle C	G ^b	187.75
1 below middle C	G	198
1 below middle C	A ^b	211.2
1 below middle C	A	220
1 below middle C	B ^b	231
1 below middle C	B	247.5
Middle C	C	264
Middle C	D ^b	281.6
Middle C	D	297
Middle C	E ^b	316.8
Middle C	E	330
Middle C	F	352

Table 5-1.
Notes and Their Frequencies (*continued*)

Octave	Note	Frequency (Hz)
Middle C	G ^b	375.5
Middle C	G	396
Middle C	A ^b	422.4
Middle C	A	440
Middle C	B ^b	462
Middle C	B	495
1 above middle C	C	528
1 above middle C	D ^b	563.2
1 above middle C	D	594
1 above middle C	E ^b	633.6
1 above middle C	E	660
1 above middle C	F	704
1 above middle C	G ^b	751
1 above middle C	G	792
1 above middle C	A ^b	844.8
1 above middle C	A	880
1 above middle C	B ^b	924
1 above middle C	B	990
2 above middle C	C	1056
2 above middle C	D ^b	1126.4
2 above middle C	D	1188
2 above middle C	E ^b	1267.2
2 above middle C	E	1320
2 above middle C	F	1408
2 above middle C	G ^b	1502
2 above middle C	G	1584
2 above middle C	A ^b	1689.6
2 above middle C	A	1760
2 above middle C	B ^b	1848
2 above middle C	B	1980
3 above middle C	C	2112
3 above middle C	D ^b	2252.8
3 above middle C	D	2376
3 above middle C	E ^b	2534.4
3 above middle C	E	2640
3 above middle C	F	2816
3 above middle C	G ^b	3004
3 above middle C	G	3168
3 above middle C	A ^b	3379.2
3 above middle C	A	3520
3 above middle C	B ^b	3696
3 above middle C	B	3960

played for the longest period of time (*duration*) is called the *whole note*. Other notes' durations are measured in fractions of the whole note. There are half, quarter, eighth, and sixteenth notes. The duration can even be less—for example, most drummers work in thirty-second and sixty-fourth notes. The duration of the whole note in any composition is pretty much up to the creator's discretion and depends on how fast the music should be played. In the section on the SOUND statement, you will read about the duration parameter, which controls the music's tempo. In the sample programs, the whole note's duration is one second. You'll see later how to represent this in your program.

Macintosh Sound

Microsoft BASIC implements sound with three functions: BEEP, SOUND, and WAVE.

BEEP creates the simple sound that alerts the user. It is very easy to use: just type **BEEP**. SOUND is the primary statement you will use to produce sound. The SOUND statement allows you to control a note's pitch, duration, and volume. If you prefer multiple-part harmony, you can assign each note to one of four voices. The SOUND WAIT and SOUND RESUME statements allow you to synchronize these voices. WAVE lets you control the quality of the sound by specifying the shape of the wave form. Musicians call this aspect of sound its *color* or *timbre*. By changing the shape of the wave form, you can produce thin, shrill sounds or rich, melodious tones.

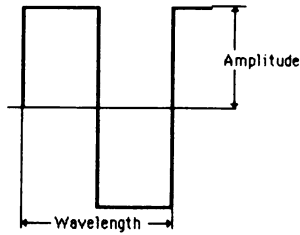
THE BEEP STATEMENT

The Macintosh sound synthesizer transforms digital information into analog waves to create sound. The simplest kind of sound is created with the BEEP statement. As the name implies, this statement emits a short beeping sound. The pitch is D above middle C. It is convenient for alerting the user to some occurrence, such as an input error. This listing shows how easy it is to use:

```
Loop:
BEEP
Skip: IF INKEY$="" THEN Skip ELSE Loop
```

Press the SPACE BAR to repeat the sound. Use Stop from the Run menu to quit the program.

BEEP uses a very simple wave form, called a *square wave*, to create sound:



A square wave is created by a signal that toggles between two states instantly, instead of following the gradual change of a sine wave. Note that although the wave is a different shape, the concepts of wavelength and amplitude still apply.

THE SOUND STATEMENT

The **SOUND** statement also uses a square wave tone, but it gives you control of the pitch (frequency), the length of the note (duration), and the volume. The format for this statement is **SOUND frequency, duration, volume, voice**. The last two parameters are optional. You can use frequency numbers from 0 to as high as you can stand listening to. The Macintosh can produce sounds ranging from 30 Hz to 11,000 Hz. BASIC allows you to specify any number for the frequency parameter; however, best results are obtained in the range from 30 to about 6000 Hz. This range encompasses the fundamental frequencies of all but the highest notes of some high-pitched wind instruments. Table 5-1 shows the frequency values for these notes.

As you learned earlier, the duration determines the length of time that the note is sustained. It can be represented by any number from 0 to 77. Since one second is about 18.2 on this scale, the longest duration you can specify with one **SOUND** statement is about 4.23 seconds.

SIMPLE SOUNDS

Let's try the **SOUND** statement, using just the frequency and duration parameters:

```
Play:
CLS
RESTORE SongData
FOR i=1 TO 12
  READ f,d
  SOUND f,d
NEXT i
PRINT "Press <space bar> for another round"
Stay: IF INKEY$="" THEN Stay ELSE Play
```

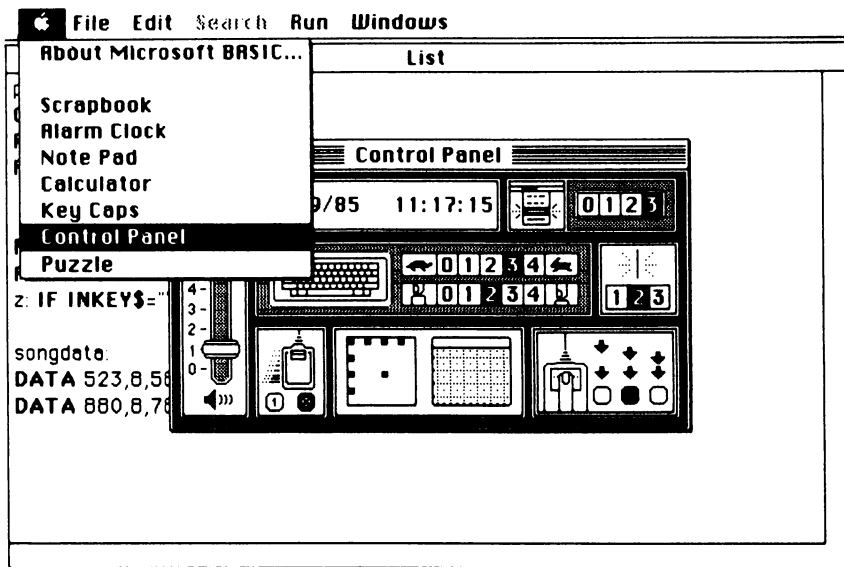
SongData:

DATA 523,8,587,8,659,8,523,8,784,4,659,8

DATA 880,8,784,8,1046,16,784,8,659,8,523,8

The note sequence in this well-known tune is C,D,E,C,G,E,A,G,C, G,E,C, where the next-to-last C is one octave above middle C. The frequencies and durations are read into f and d and then played with the SOUND function. The loop continues for the 12 pairs of numbers. You can repeat the sequence by pressing the SPACE BAR. The RESTORE statement restores the data pointer to the SongData line and reads in the data again and again.

The volume is affected by two factors: the SOUND statement's volume parameter and the volume slide switch on the Macintosh Control Panel. You can display this Control Panel with the Apple desk accessory menu.



Try the different volume levels from 0 to 7, playing the song each time. Then select a level that you find comfortable, and leave it there. Note that BEEP will not produce sound if the volume is set to 0.

You can also fine-tune the volume level with the volume parameter of the SOUND statement. The default (which you just heard) is 127 in a range of 0 to 255. First, try it at 0. Add 0 to the SOUND statement in the current program to match:

```
SOUND f,d,0
```

The volume level is perceptibly less. Try values up to 255. It doesn't have the dramatic effect of the volume slide switch, but changing the volume parameter does make a difference.

The voice parameter lets you produce as many as four voices to create four-part harmony (or discord, depending on your musical prowess). We will look at multiple-voice abilities later in this chapter. Save the current program with the file name Georgia for later experiments.

SINGLE-VOICE MUSIC

There are actually three different sound synthesizers in the Mac: square wave, four-tone, and free-form. Both the BEEP statement and single-voice SOUND statements use the square wave synthesizer. The sound produced by this synthesizer is a little harsh in comparison to the mellow tones produced by the four-tone synthesizer, but it is easy to use and is fine for graphics sound effects. The next program shows a few examples of the different kinds of sounds you can make with the square wave synthesizer. Press any key to stop a sound, and then click to select another one.

```
Setup:  
DEFINT a-z  
DIM ww(256)  
WINDOW 1,"", (20,30)-(490,330),4  
TEXTFONT 0 : TEXTSIZE 24  
MOVETO 150,40 : PRINT "Sound Maker" : TEXTSIZE 12
```

```
Buttons:  
FOR b=1 TO 6  
  READ b$ :  
  BUTTON b,1,b$,(170,60+b*30)-(390,80+b*30),3  
NEXT
```

DATA Space Ship, Alien Laser Zap, Engine, Warning Signal
DATA Close Encounters, Goodbye

Select:

WHILE DIALOG(0)<>1 : **WEND**

Selection = DIALOG(1)

ON Selection **GOSUB** SpaceShip, LaserZap, Engine, WarningSignal, Encounters, Goodbye

SOUND 0,18

GOTO Select

SpaceShip:

x=0

WHILE INKEY\$="" **AND** x<140

z=x*2 : x=x +1

FOR j=z+100 **TO** z **STEP** -25

SOUND j,.49 : **SOUND** j+200,.49

NEXT

WEND

RETURN

LaserZap:

FOR x=0 **TO** 2

SOUND 1500*RND(1),1.5,30

NEXT

RETURN

Engine:

pedal=2 : press=0

WHILE INKEY\$=""

IF 16*RND(1) <1 **THEN** press=NOT(press)

IF press=0 **THEN** pedal=pedal-.05 **ELSE** pedal=pedal+.05

IF pedal<.44 **THEN** pedal=.5 **ELSE IF** pedal >2 **THEN** pedal=2

SOUND 1,pedal

WEND

RETURN

WarningSignal:

FOR i=1 **TO** 6

SOUND 2112,3

SOUND 66,3

NEXT i

RETURN

Encounters:

RESTORE Notes

FOR i=1 **TO** 5

READ freq,dur

SOUND freq,dur

NEXT**RETURN**

Notes:

DATA 784,9,880,9,698,9,349,9,523,19

Goodbye:

SOUND 528,4**SOUND 528,4****SOUND 528,3****SOUND 396,3****SOUND 440,3****SOUND 352,4****FOR i=1 TO 5000: NEXT i****WINDOW CLOSE 1****STOP**

There are lots of things you can do with the SOUND statement. Use the following suggestions as a guide, and don't be bashful about experimenting.

Mixing Sound and Graphics

Single-voice sound can easily be added to a static graphics display, but the real trick is to coordinate SOUND with an active display. In the next program, the Olympic theme was added to the program from Chapter 3 that displays the Olympic rings with a flashing title:

```
n=16
DIM f(n),d(n)
FOR i=1 TO n
  READ f(i),d(i)
NEXT i
DATA 330,16,396,12,396,4,264,8,297,8,330,16
DATA 297,8,297,8,297,8,330,4,297,4,264,4,297,4
DATA 330,4,264,4,297,8
```

```
'draw screen
LINE(0,0)-(500,300),,bf
CIRCLE(120,80),50,30
CIRCLE(240,80),50,30
CIRCLE(360,80),50,30
CIRCLE(180,120),50,30
CIRCLE(300,120),50,30
LINE(110,200)-(374,240),30,bf
MOVETO 160,230
TEXTSIZE 18
PRINT"XXIIIrd Olympiad";
TEXTSIZE 12
```

```
r%(1)=201: r%(2)=111: r%(3)=240: r%(4)=374
```

```
TimerOn:
```

```
ON TIMER(3) GOSUB Invert: TIMER ON
```

```
Play:
```

```
FOR i=1 TO n
```

```
    SOUND f(i),d(i)
```

```
NEXT i
```

```
SOUND 0,8
```

```
FOR i=1 TO n-2
```

```
    SOUND f(i),d(i)
```

```
NEXT i
```

```
SOUND f(n),4
```

```
SOUND f(n-1),8
```

```
SOUND 0,8
```

```
GOTO Play
```

```
Invert:
```

```
    INVERTRECT VARPTR(r%(1))
```

```
RETURN
```

To alternate between graphics and sound, the statement `ON TIMER(3) GOSUB` was used. This statement interrupts the Play routine every three seconds to invert the rectangle containing the title.

There are two other features of this program that you should notice. The frequency and duration values are read into arrays `f(i)` and `d(i)` at the beginning of the program and are used in the Play routine later. In previous programs, the sounds were played as the data was read in. Storing data in arrays allows you to replay portions of the selection without duplicating data unnecessarily. In this program, the two passes use exactly the same notes, except that the last two notes are reversed.

You may also notice that inserting a pause between repetitions of the melody is not a trivial problem. A simple time delay does not always do the trick, because the Mac SOUND synthesizer stores the sound in a buffer area and then feeds the buffer to the sound port as necessary. BASIC has no way to detect when this buffer is empty and hence does not know when to start the delay. Thus, you can only guess how long to delay. One workable solution is to include a `SOUND` statement with the frequency set to zero and the duration set for the time you want to pause (`SOUND 0,8` in this program). With this technique, no sound will be produced.

One of the advantages of the simple square wave synthesizer is that it uses only 2 percent of the processor's time, so it can be combined with graphics without a significant effect on display speed. The four-tone synthesizer, on the other hand, demands 50 percent or

more of the processor's attention. This can be deadly to active graphics displays. If you're a graphics programmer, you will have to balance between sound quality and graphics speed.

The next program shows more examples of linking sound with graphics.

```
RANDOMIZE TIMER
cx=245: cy=126: r=30
GetDur:
INPUT "Enter duration (0-20)",duration
IF duration<0 OR duration>20 THEN GetDur
MOVETO cx,cy
Loop:
x=4-8*RND(1): y=4-8*RND(1)
FOR n=1 TO 3+8*RND(1)
  cx=cx+x: cy=cy+y
  CIRCLE (cx,cy),r
  SOUND 440,duration,0
  r%(0)=cy-r+1: r%(1)=cx-r+1
  r%(2)=cy+r: r%(3)=cx+r
  ERASEOVAL VARPTR(r%(0))
  SOUND 528,duration,0
NEXT n
IF INKEY$="" THEN Loop
RUN
```

Pressing the SPACE BAR runs the program again. Enter duration numbers from 1 to 20 to see how the note duration affects both the speed of the display and the response to pressing a key.

SOUND AND INTERACTIVE GRAPHICS

Sound can be used to give audible feedback to mouse or keyboard action. In the next example, sound is used to emphasize correct and incorrect cell selections in the maze program from Chapter 4. The changes are shown in the following listing:

```
'add cell to maze
SOUND 528,1

Play:
IF MOUSE(0)=0 THEN Play
x=MOUSE(3): y=MOUSE(4)
IF ABS(x-250)>w*c/2 OR ABS(y-150)>w*r/2 THEN GOSUB BadSound: GOTO Play

PaintNEWRect:
IF flag=0 THEN GOSUB BadSound
IF flag=1 THEN LINE(lx+3,ly+3)-(lx+w-3,ly+w-3),,bf: BEEP
```

```

' ...
GOTO Play

BadSound:
  IF MOUSE(0)<0 THEN SOUND 33,3
RETURN

Win:
p(1)=0: p(2)=0: p(3)=342: p(4)=512
ON TIMER (1) GOSUB Flash: TIMER ON
SOUND 352,12
SOUND 297,8
SOUND 231,16
SOUND 297,16
SOUND 352,16
SOUND 462,32
Stay: IF INKEY$="" THEN Stay
END

Flash:
  INVERTRECT VARPTR(p(1))
RETURN

```

SOUND 528,1 announces the addition of each new cell as you create the maze.

The BadSound subroutine is added just before the Win routine. It plays a low-pitched tone whenever you click outside the maze or in an incorrect cell. GOSUB BadSound sends control to this routine. BEEP is used to announce a new cell added to the maze path.

The Win routine has been modified to play a six-note melody. The ON TIMER(1) statement causes the program to leave the SOUND statement every second to invert the title rectangle. Since the sound is stored in a buffer, this interruption does not disturb the melody. The calls to the Flash routine continue until a key is pressed to end the program.

The next program shows another example of using sound to reinforce behavior. In this modified version of the Towers of Hanoi program from Chapter 4, you hear a pleasant sound when you pick a top disk and a valid tower. You hear a buzzer-like sound when you select the tower the selected disk is already on or when you select a disk that's not on the top of a stack. Lower frequencies, such as 50 and 45, give the buzzer effect. Simple sounds like this require little effort, but they add a lot to an interactive program. The changes are listed below:

```

AbortDisc:
SOUND 50,10
GOSUB GetDisc

Valid:
  NewTower = Picked
  SOUND 2000,3
RETURN

GetDisc:
  Msg$ = " Select A Disc"
  .....
  IF Disc(Tower(DiskToMove),Level(DiskToMove)-1)=0 THEN Okay = True ELSE
                                SOUND 45,5
  WEND

Leave:
  SOUND 1546,3
RETURN

```

The WAVE Statement

The WAVE statement makes two important contributions to your ability to produce sound with Microsoft BASIC. It activates and deactivates multiple-voice mode, and it allows you to change from a square wave into a wave form that provides a much more pleasant sound.

Load the program you saved earlier as Georgia; then make these changes:

```

WAVE 0,SIN
Play:
...
SOUND f,d,,0

```

What a difference! The latest version sounds much better than the original. There are two changes: the addition of the line **WAVE 0,SIN** and the addition of the wave parameter at the end of the **SOUND** statement. In the **WAVE** statement, 0 specifies voice 0 of four voices. This number corresponds to the wave parameter in the **SOUND** statement. **SIN** stands for the sine function (shown in Figure 5-1). It produces a rich, smooth sound. The default **SOUND** function (without a voice selection) uses the square wave generator. This is

fine for buzzers and simple sounds but is not appropriate for music.

There is no need to include a number for the volume parameter, since it is ignored when the multi-voice mode is active. You must still leave a place for it with two commas, however. The 0 in the SOUND statement stands for voice 0.

MULTIPLE VOICES

Speaking of voices, the next program illustrates how to create multiple-part harmony:

```

DEFINT a-z: DEFSGN d,f
FOR i=1 TO 8: READ freq(i): NEXT i
REM rest C   D   Eb F   G   A   Bb
DATA 0, 33,37.125,39.6,44,49.5,55,57.75
d=56
note$="zdefgeb"
DATA b38b38c48d48,f38f38f38f38,d38z18z14,b28z18b18z18
DATA b38d48c48f38,f38f38A38f38,d38z18E34,b28z18f24
DATA b38b38c48d48,f38f38f38f38,d38z18z14,b28z18b18z18
DATA b34e38f38,f34f38f38,d34c38z18, b24f28z18
DATA b38b38c48d48,f38f38f38f38,d38z18z14,b28z18b18z18
DATA e48d48c48b38,g38g38g38g38,b28z18z14,e28z18e18z18
DATA e38f38g38e38,e38e38e38e38,c38z18z14,f28z18f18z18
DATA b34b34,d34d34,f24f24,b24b24
DATA done
FOR i=0 TO 3: WAVE i,SIN: NEXT i

Play:
SOUND WAIT
FOR voice=0 TO 3
  READ s$
  IF s$="done" THEN Stay
  FOR k=1 TO LEN(s$) STEP 3
    f=freq(INSTR(note$,MID$(s$,k,1))) * 2 ^ (VAL(MID$(s$,k+1,1)))
    d1= d/VAL(MID$(s$,k+2,1))
    SOUND f,d1,,voice
  NEXT k
NEXT voice
SOUND RESUME
GOTO Play
Stay: IF INKEY$="" THEN Stay

```

The main lesson in this program is the way multiple voices are handled in the Play routine. This routine reads in a string containing frequency and duration information from one measure of music for a single voice. The string is analyzed, and the notes are sent via the

SOUND statement to the sound buffer. Then a string is read in for the next voice, and so on, until one measure of notes has been processed for all four voices.

Now here's the trick. The SOUND WAIT statement at the beginning of the loop causes all sound information to remain in the buffer until the entire loop is complete. Then the SOUND RESUME statement turns all four voices loose at the same time. That's how you get simultaneous multiple voices. Each line of data is processed in the same way until the piece is finished.

Take a closer look at the data. The coding scheme used in this program requires three items for each note: the note (a-g), the octave number (0-7), and the duration (1=whole, 2=half, 4=quarter, 6=sixth, 8=eighth). Every data line contains one string for each of the four voices. In the first data line, b38 means that note *b* in octave 3 is an eighth note. In that same string, there is another *b*, a *c* in the next octave up (octave 4), and a *d* also in octave 4. In the last two strings of the first data line, *z* is used as a rest.

The frequency numbers for each octave are calculated by multiplying the corresponding note in a base octave by an appropriate power of two. The base octave (three octaves below middle C) is read into the array freq. To avoid special notations for sharps or flats, these notes are coded right into the array. This approach has the disadvantage that no new sharps or flats can be introduced as the piece progresses. For this particular song, *e* is used for *E^b* and *b* is used for *B^b*.

The freq array is closely associated with the string note\$. The first character in the string is *z*; its frequency (0) is the first number in the array freq. The second character is *c*, and its frequency (33) is the second number in the array. This correspondence is used by the Play routine to translate a letter in the data line to its corresponding numeric frequency. The INSTR function takes care of all the details.

The duration of each note is calculated by dividing the number *d*=56 by the duration number in the data string. Using smaller numbers will increase the tempo of the piece. It will also demonstrate how much the multiple-voice sound synthesizer slows down program execution. Using smaller numbers will cause pauses between each measure. This delay occurs because the program cannot feed the data to the buffer before it is emptied. To speed up the tempo while maintaining continuous music would require either tighter code or machine language routines to speed up execution.

Each voice is assigned a sine wave as its wave form in a loop just above the Play routine. This is the default wave form for multiple-

voice sound. You will see how to define your own wave forms shortly.

One more observation: the volume is greatly affected by the octave you select for this song. If you choose to play it in the next octave down, the song will barely be audible, even with the volume slide switch set to 7. You can try this yourself by changing the exponent portion of the frequency calculation to $2^{(\text{VAL}(\text{MID}\$(s\$,k+1,1)) - 1)}$. The -1 shifts everything down one octave and nearly wipes out the volume at the same time.

SIMULATING MUSICAL INSTRUMENTS

Just how much control does the `WAVE` statement give you over the quality of the sound? Can you make the Mac talk, or sound like a clarinet or a trumpet? It can be done—but not without a great deal of effort. While the Mac can generate a reasonably consistent wave form for all frequencies, the wave forms generated by real musical instruments vary with the frequency. Also, the amplitude of a vibrating string, for example, builds up to a peak and then gradually decreases as the vibrations damp out. Simulating these effects on a microcomputer would require considerable knowledge and effort. Some dedicated users have achieved success by storing preselected wave patterns in different arrays and using them as needed to create the desired effect. Without this kind of effort, all the sounds you produce will sound more or less like an organ at different stops.

DEFINING YOUR OWN WAVE FORMS

So far, the examples in this book have used either the default square wave or the sine wave to generate sound. The `WAVE` statement allows you to define your own wave form, which lets you experiment with unusual sounds. The wave form is stored in an integer array of 256 numbers ranging from -128 to 127 . The numbers stand for the height of the wave, at a given point in time, above or below the line around which it oscillates. The name of the array is placed in the `WAVE` statement along with the number of the first array element in the pattern.

Zero is used as the default if no starting array element number is supplied. There are several ways to generate wave form arrays. One way is to let the random number generator have a field day selecting values for you. This method is likely to produce some interesting sounds, but many of them will not be pleasant. Another approach is to write a program that records your freehand mouse doodles into an array. This is a good exercise if you want to practice the lessons from Chapter 4. One problem with freehand wave forms is that the rela-

tionship between the wave form and the resulting tone quality is not obvious. If you have a wave form that is just about what you want, it is not clear how to change it to get the right sound. Another problem is that free-form waves may produce distorted harmonic frequencies because of the sound synthesizer's sampling rate limitations.

A more controlled method is to use combinations and variations of well-known periodic functions to create new sounds as in the following program that uses the sine function:

```

DEFINT w
DIM wav(256)
a=2*3.14159/256
FOR i=0 TO 255
  wav(i)=50*(SIN(i*a)+2*SIN(3*i*a))
  PRINT i, wav(i)
  IF wav(i)>127 OR wav(i)<-128 THEN STOP
NEXT i

FOR w=1 TO 2
  IF w=1 THEN WAVE 0,SIN: PRINT "SIN function"
  IF w=2 THEN WAVE 0,wav: PRINT "wav function"
  FOR j=1 TO 10
    READ f,d
    SOUND f,d,,0
  NEXT j
  RESTORE det
  Det:
  DATA 660,8,633.6,8,660,8,633.6,8,660,8
  DATA 495,8,594,8,528,8,440,8
  DATA 0,16
  t=TIMER
  Delay: IF TIMER<t+5 THEN delay
NEXT w

```

The values stored in the array `wav` represent the height of the curve at fixed intervals. The values must be between -128 and 127 . The function used in this example is $y = 50(\sin(x) + 2\sin(3x))$. You can play with different combinations of the trigonometric functions (`SIN`, `COS`, `ATN`, and so on) or toss in a few of your own. Another interesting example you might want to try is this: $y = 80\sin(x) - 80 + i/2$.

The delay loop pauses the program between passes to let the sound buffer process its contents.

Applications

The first two applications give you two more tools for generating different wave patterns.

WAVE PLOTS

You can start out with a program that plots a graph of different functions and plays a tune so that you can hear the tone quality. The function should be entered in the Funct line. Press any key to stop the program:

```

DEFINT a-z:DEFSNG a,f,x,y
DIM wav(256)
a=2*3.14159/256
FOR i=0 TO 255
  Funct:
  wav(i)=50*(SIN(i*a)+2*SIN(3*i*a))
  PRINT i, wav(i)
  IF wav(i)>127 OR wav(i)<-128 THEN STOP
NEXT i
WAVE 0, wav

'draw axes
ON TIMER(4.7) GOSUB DoSound:TIMER ON
CLS
TEXTSIZE 10
TEXTFONT 10
TEXTMODE 9
LINE (50,130)-(450,130)
FOR x=50 TO 450 STEP 100
  IF x=250 THEN Skip1
  LINE (x,128)-(x,132)
  left=10
  IF x=50 OR x=450 THEN left=15
  MOVETO x-left,144:PRINT (x-50)/100-2,"π";
Skip1:
NEXT x
FOR y=10 TO 250 STEP 20
  IF y=130 THEN MOVETO 240,135:GOTO SkipMvve
  MOVETO 218,y+5
  SkipMvve:
  PRINT 120-(y-10);
  LINE(248,y)-(252,y)
NEXT y
LINE(250,10)-(250,250)
TEXTSIZE 12
TEXTMODE 0
TEXTFONT 1

'draw waveform
FOR m=0 TO 1
  FOR i=0 TO 255
    y=wav(i)
    IF y>130 OR y<-130 THEN Skip2
    x=i*2/255-2+m*2

```

```

x1=100*x+250
y1=130-y
PSET(x1,y1)
Skip2:
NEXT i
NEXT m
Stay: IF INKEY$="" THEN Stay
TIMER OFF:END

DoSound:
FOR j=1 TO 10
  READ f,d
  SOUND f,d,d
NEXT j
RESTORE DoSound
DATA 316.8,12,316.8,12,316.8,8,352.4,396,12
DATA 396,8,352.4,396,8,422.4,4,462,16
RETURN

```

The function-plotting routine is a modification of an earlier program. Figure 5-2 shows the results with the current function.

ON TIMER(4.7) is used to send the program to the sound subroutine about every five seconds. Be careful not to do this too often, or the sound buffer will demand too much of the processor's time and halt the screen graphics.

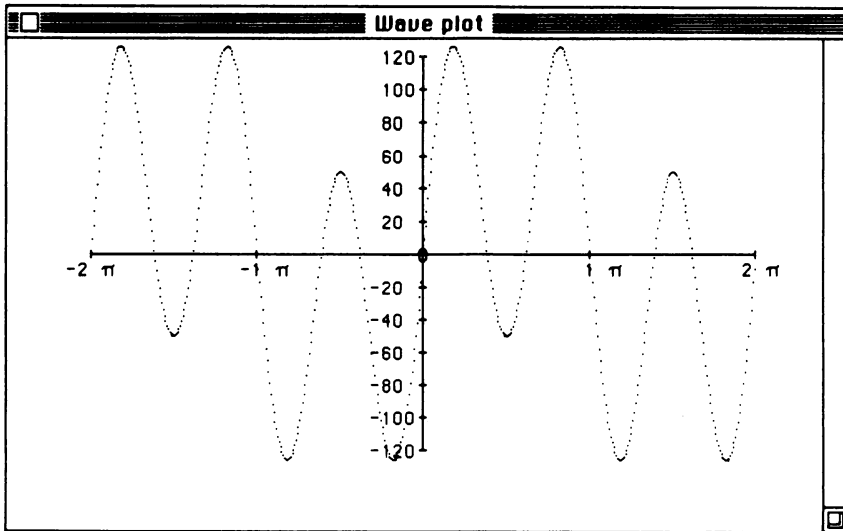


Figure 5-2.
Wave form plot

Another way to create new wave forms is by combining modified sine waves. The following program lets you define as many as ten sine waves, each with its own amplitude and phase shift. The program then adds them together to produce a totally new wave form. As in the previous program, the wave form is plotted and played.

The 256 values describing this new shape are stored in the swave array. You can modify the program to save the values to disk or print them out for future reference.

```

RANDOMIZE TIMER
DEFINT i,j,n,s
DIM swave(256)
 $c=2*3.14159/256:2\pi/256$ 

'fill array
INPUT "Enter # of wave parts (1 - 10)" n
DIM a(n),p(n)
PRINT "Enter",n,"amplitudes (0 - 127)"
FOR i=1 TO n
    lna:
    INPUT a(i)
    IF a(i)<0 OR a(i)>127 THEN BEEP:GOTO lna
NEXT i
PRINT "Enter",n,"phase values (0 - 6.28)"
FOR i=1 TO n
    lnp:
    INPUT p(i)
    IF p(i)<0 OR p(i)>6.28 THEN BEEP:GOTO lnp
NEXT i
LOCATE 1,30:PRINT "Filling array."
FOR i=0 TO 255
    w=0
    FOR j=1 TO n
         $w=w+a(j)*\text{SIN}(c*i*j+p(j))$ 
    NEXT j
     $\text{swave}(i)=w/n$ 
    LOCATE 2,29:PRINT i,swave(i);
NEXT i
WAVE 0,swave

'draw axes
ON TIMER(5) GOSUB DoSound:TIMER ON
CLS
TEXTSIZE 10
TEXTFONT 10
TEXTMODE 9
LINE (50,130)-(450,130)
FOR x=50 TO 450 STEP 100
    IF x=250 THEN Skip
    LINE (x,128)-(x,132)

```

```

left=10
IF x<50 OR x>450 THEN left=15
MOVETO x-left,144:PRINT (x-50)/100-2;"π"
Skip:
NEXT x
FOR y=10 TO 250 STEP 20
  IF y=130 THEN MOVETO 240,135:GOTO SkipMove
  MOVETO 218,μ+5
  SkipMove:
  PRINT 120-(y-10);
  FOR x=248 TO 252:PSET(x,μ):NEXT x
NEXT y
FOR y=10 TO 250:PSET(250,μ):NEXT y
TEXTSIZE 12
TEXTMODE 0
TEXTFONT 1

'draw waveform
FOR m=0 TO 1
  FOR i=0 TO 255
    y=swave(i)
    IF y>130 OR y<-130 THEN Skip2
    x=i*2/255-2+m*2
    x1=100*x+250
    y1=130-y
    PSET(x1,μ1)
  Skip2:
  NEXT i
NEXT m
Stay:IF INKEY$="" THEN Stay
TIMER OFF:RUN

DoSound:
SOUND 784,16,0
SOUND 880,16,0
SOUND 698,16,0
SOUND 349,16,0
SOUND 523,24,0
SOUND 0,16
RETURN

```

The function you create is represented by the formula

$$F(X) = A1*\text{SIN}(P1+X) + A2*\text{SIN}(P2+X) + A3*\text{SIN}(P3+X) + \dots + An*\text{SIN}(Pn+X)$$

where A1 through An represent the amplitudes you enter, and P1 through Pn represent the phase shift values. These functions can be used to change the wave form in other programs.

KEYBOARD SIMULATION

Here's a program that turns part of your computer keyboard into a piano keyboard:

```
'initialize variables
DEFINT a-z: DEFMSG f
duration=5
TEXTMODE 3
TEXTSIZE 14
TEXTFACE 1
DIM key$(16,2), freq(16), rect(64),poly(291), pat(8)
DIM index(16)

FOR i=1 TO 16: READ key$(i,1): NEXT i
FOR i=1 TO 16: READ key$(i,2): NEXT i
DATA z,s,x,c,f,v,g,b,n,j,m,k,"",l,./
DATA A,Bb,B,C,Db,D,Eb,E,F,Gb,G,Ab,A,Bb,B,C

'read in frequencies
FOR i=1 TO 16: READ freq(i): NEXT i
DATA 220,231,247.5,264,281.6,297,316.8,330
DATA 352,375.5,396,422.4,440,462,495,528

'draw keyboard
LINE(100,20)-(400,220),,b
FOR i=100 TO 400 STEP 30
  LINE(i,220)-(i,20)
  IF i=130 OR i=190 OR i=280 THEN LINE(i-13,20)-(i+7,160),,bf
  LINE(387,20)-(400,160),,bf
  IF i=310 THEN LINE(i-10,20)-(i+10,160),,bf
  IF i=220 OR i=340 THEN LINE(i-7,20)-(i+13,160),,bf
NEXT i

'set up polygon array
FOR i=0 TO 291: READ poly(i): NEXT i
DATA 38,21,101,220,130,220,101,220,130,161,130,161
DATA 117,21,117,21,101,220,101,30,21,118,160,137,160
DATA 118,160,137,21,137,21,118,160,118,38,21,131,220
DATA 160,220,131,220,160,21,160,21,138,161,138,161
DATA 131,220,131,38,21,161,220,190,220,161,220,190
DATA 161,190,161,177,21,177,21,161,220,161,30,21,178
DATA 160,197,160,178,160,197,21,197,21,178,160,178
DATA 46,21,191,220,220,220,191,220,220,161,220,161
DATA 213,21,213,21,198,161,198,161,191,220,191,30,21
DATA 214,160,233,160,214,160,233,21,233,21,214,160,214
DATA 38,21,221,220,250,220,221,220,250,21,250,21,234
DATA 161,234,161,221,220,221,38,21,251,220,280,220,251
DATA 220,280,161,280,161,267,21,267,21,251,220,251
DATA 30,21,268,160,287,160,267,160,287,21,287,21,268
DATA 160,268,46,21,281,220,310,220,281,220,310,161
DATA 310,161,300,21,300,21,288,161,288,161,281,220,281
```

```

DATA 30,21,301,160,320,160,301,160,320,21,320,21,301
DATA 160,301,46,21,311,220,340,220,311,220,340,161
DATA 340,161,333,21,333,21,321,161,321,161,311,220,311
DATA 30,21,334,160,353,160,334,160,353,21,353,21,334
DATA 160,334,38,21,341,220,370,220,341,220,370,21,370
DATA 21,354,161,354,161,341,220,341,38,21,371,220,400
DATA 220,371,220,400,161,400,161,387,21,387,21,371
DATA 220,371

```

```

'index to polygon array
FOR k= 1 TO 16: READ index(k): NEXT k
DATA 0,19,34,53,72,87,110,125,144,163,178,201,216,239
DATA 254,273

```

```

'pattern array
FOR l=0 TO 7: READ pat(l): NEXT l
DATA -21931,-21931,-21931,-21931,0,0,0,0

```

```

Scan:
in$=INKEY$
IF in$="" THEN Scan
IF in$=" " THEN Halt
FOR k=1 TO 16
  IF in$=key$(k,1) THEN Play
NEXT k
GOTO Scan

```

```

Play:
SOUND freq(k),duration
FILLPOLY VARPTR(poly(index(k))), VARPTR(pat(0))
PRINT key$(k,2);
FOR i=1 TO 1000: NEXT i
IF k=2 OR k=5 OR k=7 OR k=10 OR k=12 OR k=14 THEN
  GOSUB Black ELSE GOSUB White
GOTO Scan

```

```

Black:
  PAINTPOLY VARPTR(poly(index(k)))
RETURN

```

```

White:
  ERASEPOLY VARPTR(poly(index(k)))
RETURN

```

```

Halt:
TEXTMODE 0: TEXTSIZE 12: TEXTFACE 0
END

```

It stores the active keys and corresponding notes in the two-dimensional array key\$. The INKEY\$ function detects a keystroke and then sends control to the Play routine. The corresponding fre-

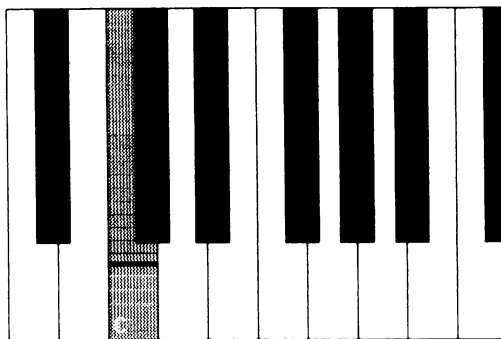


Figure 5-3.
Piano keyboard

quency is played for the time set by the variable duration. The polygon corresponding to that note is then filled with a pattern for a short period. The major chunk of the data consists of numbers that define the polygons. Figure 5-3 illustrates the program in action. The bottom two rows of letters on the Mac keyboard are assigned keys on the piano keyboard, as shown in Figure 5-4.

MUSIC ENTRY SYSTEM

The SOUND and WAVE statements are excellent for creating sound, but if you are trying to translate a written piece of music, you must rely on a translation table like Table 5-1. Why can't you simply enter notes as letters and teach the computer to translate for you?

That's exactly what was done in the program that plays multiple-voice music. Although that program was not fancy, it got the job done. Microsoft has taken this idea one step further. One of the demonstration programs included with Microsoft BASIC (MUSIC) has a more sophisticated coding system. It reads music that has been coded as DATA statements for as many as four voices and plays the music for you. A modified version is shown here:

```
DEFINT A-Z
DIM F%(88)
Log2of27.5% = LOG(27.5%)/LOG(2%)
FOR x%=1 TO 88
```

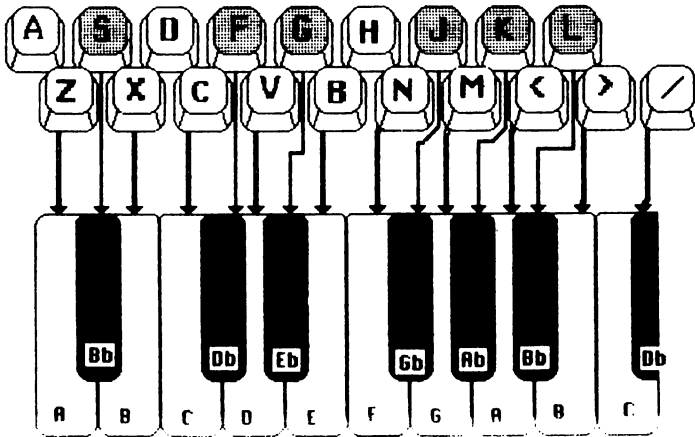



Figure 5-4.
Note assignments

```

F#(x%) = 2^(Log2of27.5# + x%/12#)
NEXT x%

'Build fundamental wave form
PRINT "use SIN wave (y/n)?";
Stay: i$=INKEY$: IF i$="" THEN Stay
PRINT i$
IF i$="y" OR i$="Y" THEN Sine
PRINT "Building array:";
DIM Timbre(255)
K# = 2*3.14159265# / 256
FOR I=0 TO 255: LOCATE 2,14: PRINT i
  Timbre(I) = 31*(SIN(1*K#) + SIN(2*1*K#) + SIN(3*1*K#) + SIN(4*1*K#))
NEXT I
WAVE 0, Timbre
WAVE 1, Timbre
WAVE 2, Timbre
WAVE 3, Timbre
GOTO SymbolTable

Sine:
WAVE 0, SIN
WAVE 1, SIN
WAVE 2, SIN
WAVE 3, SIN

```

SymbolTable:

C\$ = "cdefgebp"-123468<>1"

DIM CF(19)

FOR I=1 TO 19

 READ CF(I)

NEXT I

DATA 0,2,4,5,7,9,11,0,1,-1, 0,0,0,0,0,0, -12,12,0

'Duration Values

DIM CT\$(19)

FOR I=1 TO 19

 READ CT\$(I)

NEXT I

REM--- p1,p2,p3,p4,p6,p8 correspond to 36.4...455 time units

DATA 0,0,0,0,0,0,0,0,0,36.4,18.2,12.1333333,9.1,6.0666667\$,455,0,0,0

RePlay:

SOUND RESUME

RESTORE Song

' read default octaves for voices 0-3

FOR v=0 TO 3

 READ VO(v)

 VO(v)=12*VO(v) + 3

NEXT v

Loop:

SOUND WAIT

FOR v=0 TO 3

 t\$=VT\$(v)

 Fi=-1

 READ p\$

 IF p\$="x" THEN RePlay

FOR i=1 TO LEN(p\$)

 Ci=INSTR(C\$,MID\$(p\$,i,1))

 IF Ci>8 THEN NoSound

 IF Fi>=0 THEN SOUND F\$(Fi),t\$,,v: t\$=VT\$(v)

 IF Ci=8 THEN Fi=0 ELSE Fi=CF(Ci)+VO(v)

 GOTO Skip

NoSound:

 IF Ci<11 THEN Fi=Fi+CF(Ci): GOTO Skip 'e or -

 IF Ci<17 THEN t\$=CT\$(Ci): GOTO Skip '1 through 8

 IF Ci<19 THEN VO(v)=VO(v)+CF(Ci): GOTO Skip 'x or >

 i=i+1 'ln

 VT\$(v)=CT\$(INSTR(C\$,MID\$(p\$,i,1)))

 IF Fi<0 THEN t\$=VT\$(v)

 Skip:

NEXT i

```

IF Fi>=0 THEN SOUND F*(Fi),t*,v
NEXT v
SOUND RESUME
GOTO Loop

```

Song:

```

DATA 1,3,3,3
DATA l2g>ge, l2p2de, l2p2l6g3f#g3a, l6p6gab>dcced
DATA <b>e<e, ge<b, b3ab3ge3d, dgf#g<bgab
DATA ab>c, a>dc, e3f#g3de3<b, >cdedc<babg
DATA df#d, c<a>f#, a3>da3ga3f#, f#gadf#a>c<ba
DATA gec, g<g>e, d3f#g3f#g3a, bgab>dcced
DATA <b>ed, ge<b, b3ab3ge3g, dgf#g<bgab
DATA cc#d, >ced, a3f#g3e<a3>c, e>dc<bagdgf#
DATA <gp, dp, <b3p, gp
DATA x

```

To use the program, you type notes and pauses directly from the sheet music into DATA statements. You can use BASIC's editing features to edit the data as desired. The first four data numbers select the starting octave for each voice. The octaves range from 0 to 7, 3 being the octave from middle C to B above middle C.

The rest of the data is the music. Each line in the original music program feeds the equivalent of about 1 1/2 seconds of sound per data line for each of voices 0 through 3. If you add more, the sound buffer may run out of memory; if you add less, the song will be filled with pauses.

The notation itself is very straightforward. The notes a through g can be followed by the symbols # for sharp and – for flat, and by a duration number. The duration numbers may be 1 for a whole note, 2 for half, 4 for quarter, 6 for sixth, and 8 for eighth. The duration for the notes in each voice is set with the letter l followed by a duration number. You set the basic duration in the first measure and then change it only as needed. To cause a rest for a voice, use p followed by a duration number. The seven symbols a through g refer to notes in the current octave. To play notes in an adjacent octave, use < to move down an octave and > to move up an octave.

Once you have entered the data for your song, the program plays it in an endless loop. The RESTORE statement (see RePlay:) restores the data pointer to the line Song, and the beat goes on.

You can use your own wave forms to modify the sound quality, or you can keep the standard SIN wave.

Summary

As you have seen from the examples, sound adds an exciting dimension to graphics displays. It gives your programs the professional touch that separates the mediocre program from the outstanding one.

There is no “best” sound to use for all situations. You can use BEEP or the simple square wave SOUND statement. Or you can produce fuller sounds with the WAVE statement. Which statement you use and when you use it is up to you.

BASIC Statements

BEEP
SOUND
TIMER
TIMER ON
WAVE

6

Transferring Images

The next two chapters are closely related. In this chapter you will learn how to collect and display large screen images. In Chapter 7, you will put these techniques to use by producing animation.

So far, we've used BASIC statements and ROM calls to manipulate text and to draw different graphics shapes—everything from points to rounded rectangles and polygons. In this chapter you will learn how to collect these images, as well as images from other programs. You will also learn how to save them to disk, load them back into memory, and redisplay them almost instantaneously anywhere on the screen or the printer, expanded or compressed to your specifications. These are the essential techniques for manipulating images quickly on the Mac. Mastering these techniques paves the way for animation.

There are two primary ways of storing graphics in memory. One way is to store data as a *bit map*, a bit-for-bit representation of the screen pixels that is stored in an integer array. The other method is to store the data as a *picture*, a series of drawing commands stored in a string variable. We will examine the techniques and effects of both of these methods.

Storing Images in Arrays (Bit Maps)

The array storage method uses the GET and PUT statements. GET takes information from the screen and stores the bit map into memory; PUT transfers the bit map from memory onto the screen.

BIT MAP STORAGE IN MEMORY

The storage format used by GET and PUT is straightforward: BASIC stores the image as a series of integers. When you declare an array to hold a bit map, you should declare it as an integer. If you use other data types, you will not be able to interpret the data properly.

Figure 6-1 gives an example of how an image is stored in memory. The first cell of the integer array, `a(0)`, contains the width of the rectangle; the second cell, `a(1)`, contains the height of the rectangle. The remaining cells contain the bit pattern of the image. The bits are arranged in groups of 16 horizontally. For this example, the pattern is 21 bits wide, so a 32-bit grid is used. When the pattern is displayed, the rightmost eleven bits of each line are ignored, since the computer can tell from the width value in `a(0)` that it should display only the first 21 bits. In the figure's example, cells `a(2)` and `a(3)` contain the bit pattern for the first horizontal line. Cell `a(2)` is the left side of the line, and cell `a(3)` is the right side. The remaining cells continue left to right and down the bit map.

In the example, the first cell is exploded to show you how the integer value in the cell is calculated. Reading from right to left, you can see that each bit in the cell represents an increasing power of 2, as shown below each bit. The exception is the leftmost bit, the "sign bit." To calculate the cell's value, you add the values of the bits that are on (black). If the sign bit is on, subtract 32768, giving the proper negative value. In our example, the first cell is calculated by summing 4, 8, 16, 32, 64, 128, and 256, giving the value 508.

As you can see, creating a drawing and converting it to a bit map can be a long, tedious process. This is the way most images were created on earlier microcomputers that produced bit-mapped graph-

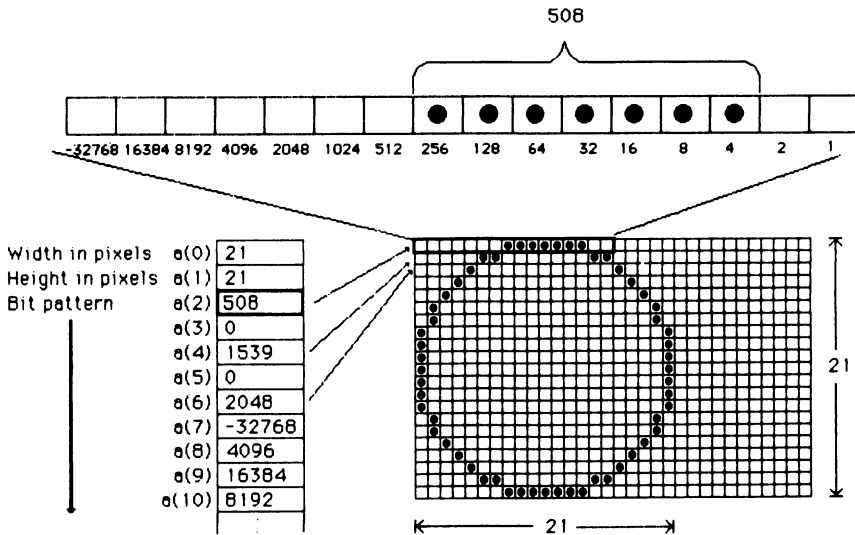


Figure 6-1.
 Array storage format

ics. The task is simple and fast on a Macintosh, thanks to a utility called Paint Mover. It converts drawings made with MacPaint into the proper bit-map data format that BASIC needs. (We'll look at Paint Mover again later in this chapter.)

TRANSFERRING A SCREEN IMAGE TO AN ARRAY (GET)

To see how the GET command works, enter this program.

```
DIM hold%(708)
FOR x = 0 TO 100 STEP 100
  FOR y = 0 TO 100 STEP 100
    FOR i = 0 TO 100 STEP 10
      IF y = 100 THEN LINE (x,y)-(i,0) ELSE LINE (x,y)-(i, 100)
      IF x = 100 THEN LINE (x,y)-(0, i) ELSE LINE (x,y)-(100,i)
    NEXT i
  NEXT y
NEXT x
GET (0,0)-(100,100),hold%
```

The program draws a figure on the screen and then, using the GET command, loads the image into an array called hold%. In this example, the image is drawn within a rectangle that is bounded by the top left coordinate (0,0) and the bottom right coordinate (100, 100). These coordinates are used in the GET command to tell BASIC which portion of the screen should be retrieved.

The size of the array hold% is calculated with the following formula:

$$((\text{bottom} - \text{top} + 1) * \text{INT}(((\text{right} - \text{left} + 16) / 16)) + 1)$$

In the formula, bottom is the highest Y coordinate (100) and top is the lowest (0). Right is the highest X coordinate (100) and left is the lowest (0). Applying the values in the example to the formula, you get

$$((100 - 0 + 1) * \text{INT}(((100 - 0 + 16) / 16)) + 1 = 708)$$

TRANSFERRING ARRAY DATA TO THE SCREEN (PUT)

To display the saved image on the screen, add the following lines to the end of the program you just entered:

```
CLS
FOR i = 1 TO 10
  PUT (RND*400,RND*200),hold%,PSET
NEXT i
```

The new code clears the screen and then displays ten copies of the image at random locations on the screen. The coordinate given in the PUT command tells BASIC the location of the upper-left corner of the image. The image is displayed to the right and below the point specified. Following the coordinate is the name of the array holding the image. In the example, we specified hold%. If no subscript is specified, BASIC starts from the beginning of the array.

The last portion of the PUT statement is the display option. The PUT command has different options that determine how an image interacts with the current screen contents. The option PSET was used in this example. The other options are PRESET, AND, OR, and the default XOR. Figure 6-2 shows how each of the options affects three different backgrounds: white, gray, and black. Try these options for yourself by changing the PUT command in the example.

The PSET and XOR options are particularly useful for animating screen images. PSET will clean up leftover points as an object moves

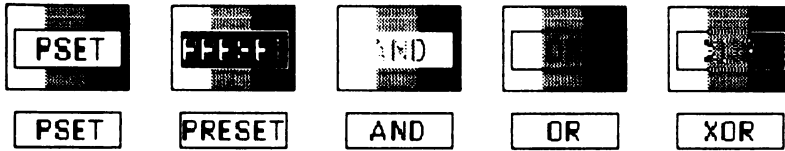


Figure 6-2.
PUT options

around the screen. XOR preserves the screen background as an object moves over it. Chapter 7 will discuss both options in detail.

CHANGING THE SIZE OF THE IMAGE

Besides simply specifying where to display a bit map, PUT also allows you to specify two coordinates. As in the GET command, the first coordinate specifies the top left corner of the display, and the second coordinate specifies the bottom right corner. When you define both corners, the PUT command will automatically scale the image to fit the new dimensions.

To see how this works, replace the PUT loop in the sample program with the following code:

```
FOR i = 1 TO 10
  x = RND*400
  dx = (RND*150)+50
  y = RND*200
  dy = (RND*150)+50
  PUT (x,y)-(x+dx,y+dy),hold%,PSET
NEXT i
```

This revised program will display the image ten times using random locations, widths (dx), and heights (dy).

Storing Images in Strings

The second method of storing images is with string variables. The BASIC statements and functions required include PICTURE, PICTURE ON, PICTURE OFF, and PICTURE\$.

CREATING A PICTURE STRING

Creating a picture string can be compared to using a tape recorder. You turn the recording mechanism on with **PICTURE ON**. Then, as the computer executes graphics commands, it records each command in the string **PICTURE\$**. When the drawing is complete, you turn the recording off with **PICTURE OFF**.

To show you how this works, we will use the previous program with a few changes:

```

PICTURE ON
FOR x = 0 TO 100 STEP 100
  FOR y = 0 TO 100 STEP 100
    FOR i = 0 TO 100 STEP 10
      IF y = 100 THEN LINE (x,y)-(i,0) ELSE LINE (x,y)-(i, 100)
      IF x = 100 THEN LINE (x,y)-(0, i) ELSE LINE (x,y)-(100,i)
    NEXT i
  NEXT y
NEXT x
PICTURE OFF
hold$ = PICTURE$
FOR i = 1 TO 10
  PICTURE (RND*400,RND*200),hold$
NEXT i

```

Instead of using **GET** and **PUT**, this program starts the picture-creation process by executing the **PICTURE ON** command. Next, the lines of the design are drawn. While the drawing is taking place, **BASIC** creates a coded string called **PICTURE\$** that contains a log of all drawing commands executed during the picture-creation session. The session is ended by the **PICTURE OFF** command. At that point, the string **PICTURE\$** is available for you to use.

Notice that the screen does not display the image as it is drawn unless **SHOWPEN** is included. This feature of **PICTURE ON** could be used to construct an image that would not be shown to the viewer until it was complete.

In this sample program, **hold\$** is assigned the value of **PICTURE\$**. Storing **PICTURE\$** in another string variable allows you to create many pictures in the same program. Each new picture you create can be stored in a different string.

Once the picture is created, you can display it on the screen by using the **PICTURE** command. This command looks a lot like the **PUT** command. One difference, however, is that **PICTURE** does not provide the display options that **PUT** provides. You may only display

picture strings in the same manner as they were created.

The display loop at the end of the program displays the picture at ten random locations on the screen. Notice the difference in the display. When you used PUT, the images flashed on the screen. Using the PICTURE command, you can see each individual line as it is being drawn.

CHANGING THE SIZE OF A PICTURE STRING

As with the PUT statement, you can resize an image by specifying a lower-right coordinate in the PICTURE statement. Replace the display loop at the end of the sample program with the following code:

```
FOR i = 1 TO 10
  x = RND*400
  dx = (RND*150)+50
  y = RND*200
  dy = (RND*150)+50
  PICTURE (x,y)-(x+dx,y+dy),hold$
NEXT i
```

Now run the revised program. Something is wrong! The resized images are much smaller than the ones generated by the PUT statement. This is a subtle difference between the PUT and PICTURE statements. The PUT statement resizes the image based on the original size of the image—in this case, 101 bits wide and 101 bits high. The PICTURE statement resizes the image in relation to the size of the window when the PICTURE statement was created.

To see how this works, add a STOP statement to the program immediately following the PICTURE OFF statement. Then make the output window smaller by clicking and dragging in the output window's size box. Next, run the program. The program will stop after the picture has been created. At this point, the output window is still blank. Again, using the window's size box, make the window large enough to fill the screen. Restart the program by selecting Continue from the Run menu. The images drawn are now larger than the ones from the initial run.

SHOWPEN and HIDEPEN

The BASIC commands SHOWPEN and HIDEPEN are used to make visible or to hide the results of drawing commands. When the PICTURE ON statement is executed, BASIC will automatically exe-

cute HIDEPEN, which causes the results of all drawing commands to be invisible and allows the picture to be created without affecting the screen. When the PICTURE OFF statement is executed, BASIC will also execute SHOWPEN, making drawing commands visible again.

You can control drawing visibility yourself by adding SHOWPEN and HIDEPEN statements to your program. If you wish to see a picture as it is being created, add a SHOWPEN statement after a PICTURE ON statement. This will counteract the HIDEPEN command that BASIC automatically executes. You can also add a HIDEPEN statement before the PICTURE OFF command to counteract its SHOWPEN.

Be careful when you use these commands. The effects of SHOWPEN and HIDEPEN are cumulative; that is to say, if you execute HIDEPEN three times before a SHOWPEN, you must execute SHOWPEN three times before drawings become visible again. The system contains a pen status register, which acts as a counter. When the register is zero or greater, drawings are visible. When the register is less than zero, drawings are invisible. Executing SHOWPEN adds 1 to the register; executing HIDEPEN subtracts 1 from the register. When you run BASIC, the register starts at zero (visible). If HIDEPEN is executed three times, the register will contain -3 (invisible). Executing SHOWPEN once will increment the register to -2 (still invisible). It will take two more executions of SHOWPEN to return the register to zero.

Comparing Storage Methods

Which method of storing an image in memory is the best for your needs? This question will become very important when you use images for animating objects.

Speed is one of the critical factors in creating believable animation. Which storage method can recreate an image on the screen in the least time? The answer is, "It depends."

Two important factors determine which method is most appropriate. One is the size of the image; the other is its complexity (the number of QuickDraw calls it requires). PUT slows down for large images, but the complexity doesn't matter, because the image is stored as a bit pattern in memory. PICTURE is fast if the number of QuickDraw calls is minimal, but it slows down for a complex image.

The following listing loads a large rectangle into memory in both formats. Then the array format is displayed in a loop by PUT, and the time is recorded by TIMER. The string format is displayed by PICTURE in the same size loop. The string format performs the same operation in less than half the time. Thus, for rapid display of large, simple figures, string storage is preferred.

```
'draw picture and store in memory
RANDOMIZE TIMER
DEFINT a: loop=50
PICTURE ON: SHOWPEN
FOR i=1 TO 20
  CIRCLE(245+RND*20,126+RND*20),10
NEXT i
PICTURE OFF
x1=225: y1=106: x2=265: y2=146
n=2+((y2-y1)+1)*INT(((x2-x1)+16)/16)
DIM array(n)
GET (x1,y1)-(x2,y2),array
CLS
PRINT"array and string loaded"
GOSUB Pause
```

```
'display array
start=TIMER
FOR i=1 TO loop
  PUT (225,106),array,OR
  CLS
NEXT i
done=TIMER
elapsed=done-start
PRINT "elapsed time for array: ";elapsed
GOSUB Pause
CLS
```

```
'display string
start=TIMER
FOR i=1 TO loop
  PICTURE ,PICTURE$
  CLS
NEXT i
done=TIMER
elapsed=done-start
PRINT"elapsed time for string: ";elapsed
Stay: IF INKEY$="" THEN Stay
END
```

```
Pause:
FOR i=1 TO 5000: NEXT i
RETURN
```

However, array storage is clearly the champion in this speed contest, as you have seen in the earlier examples. The PICTURE command has to redraw the image each time the image is displayed, which slows it down significantly.

PRESERVING THE BACKGROUND

Another difference that you might want to consider besides speed is how the imaging technique affects the current screen contents. In some instances you will want an object to float across the screen without affecting the background. (The mouse pointer is an example of this.)

The PICTURE statement simply writes over the background in OR mode—black dots in the picture image change the screen pixels to black; white dots don't affect the current screen contents. But there is no way to recover the original screen contents unless you stored them in memory first.

The PUT statement is good for controlling interaction with the background. The XOR mode has a unique property: if you PUT an image in XOR mode twice in a row, the PUT image disappears, leaving the original background intact. You will see this in action in Chapter 7.

CONSTRUCTING IMAGES

One advantage of string storage over array storage is that you can create images out of sight of the viewer. With array storage you must assemble the image on the screen and then GET the image into an array. The PICTURE ON statement does not show the image being created unless you issue a SHOWPEN statement.

One final advantage of string storage is that it stores images in the same format as the Clipboard. Thus, string storage is the natural choice when you transfer images between BASIC and other programs via the Clipboard. The next section shows how this is done.

Transferring Images Between BASIC and Applications Programs

Until now, the programs in this chapter created images with BASIC statements and stored them in string or array format. This section looks at techniques for using figures created by other programs, such as MacPaint or Microsoft Chart, with BASIC programs.

The focal point of the transfer facility is the Clipboard. The Clipboard stores images as a sequence of QuickDraw routines, just like those stored in PICTURE\$. Because the QuickDraw routines are stored in ROM, the Clipboard storage format provides a standard that can be used by all Macintosh programs.

You know, for example, that a MacPaint screen can be copied into the Clipboard, and that it will stay there until something else replaces it. You could draw an image with MacPaint, copy it to the Clipboard, quit MacPaint, and load BASIC. The MacPaint image would still be in the Clipboard.

You can also copy images from the Scrapbook into the Clipboard without ever leaving BASIC. Here is a quick review of the process: choose Scrapbook from the Apple menu, choose a picture, and then select Copy from the Edit menu to copy the image to the Clipboard. Finally, close the Scrapbook window.

TRANSFERRING IMAGES FROM THE CLIPBOARD TO BASIC

Use one of the methods just given to copy an image to the Clipboard. With that done, the only question is how to pull that image into a BASIC string variable. You need a program that will read the Clipboard contents and store them in a BASIC-compatible format. As you enter the following program, be careful not to use the Copy or Cut option from the Edit menu, or the image you stored in the Clipboard will be lost. Enter the following:

```
PRINT "Copy a picture from the Scrapbook"
INPUT "Hit return to continue", x
CLS
OPEN "CLIP:PICTURE" FOR INPUT AS 1
i$=INPUT$(LOF(1),1)
CLOSE 1
PICTURE ,i$
```

This program transfers an image from the Clipboard to the string variable i\$. It addresses the Clipboard as a device, just as it would address a disk file. The device name reserved for the Clipboard is CLIP:PICTURE. The program opens the Clipboard as a file for input. INPUT\$ brings the image from the Clipboard into the string variable i\$. Then the Clipboard file is closed. The PICTURE statement displays the image on the screen.

Because no coordinates were specified in the PICTURE statement, the image is displayed where it was recorded originally. To specify your own rectangle and see how PICTURE scales the image,

you can either add (x1,y1) to pick a different position or use (x1,y1)-(x2,y2).

TRANSFERRING IMAGES FROM BASIC TO THE CLIPBOARD

The next step is to learn how to transfer an image from a BASIC string variable back to the Clipboard. You can use this technique to transfer an image created in BASIC to an applications program.

The following program demonstrates how to do this. First, an image is created with PICTURE ON, so that it is recorded in the string PICTURE\$. Then the Clipboard is opened for output and the image is sent via the PRINT# statement. To verify that the image made it safely to the Clipboard, the Clipboard contents are loaded back into a string and redisplayed on the screen in a compressed form:

```
'draw and record picture
PICTURE ON: SHOWPEN
FOR t=0 TO 3.14 STEP .01
  x1=240+220*COS(t)*COS(4*SIN(2*t))
  y1=200+160*SIN(t)*COS(4*SIN(2*t))
  x2=170+220*COS(t)*COS(3*t/2)
  y2=100+80*SIN(t)*COS(3*t/2)
  LINE (x1,y1)-(x2,y2)
NEXT t
PICTURE OFF

'store image in clipboard
OPEN "CLIP:PICTURE" FOR OUTPUT AS 1
PRINT #1, PICTURE$
CLOSE 1
CLS

'recall image from clipboard and display
OPEN "CLIP:PICTURE" FOR INPUT AS 1
i$=INPUT$(LOF(1),1)
CLOSE 1
PICTURE (50,50)-(450,200),i$
Stay: IF INKEY$="" THEN Stay
```

Storing Images on Disk

Moving images in and out of BASIC is a big advantage, but wrestling with MacPaint every time you want to move the same image into BASIC can get tiresome.

A better approach is to capture the image on disk the first time it is transferred into BASIC. Then you can recall it directly from the

disk into memory without swapping disks and programs. Because images brought in through the Clipboard are stored in string format, that's where this story will start.

STRING STORAGE ON DISK

Storing a string-formatted image on disk is a two-step process: loading the image into a string, and saving the image to disk. The next program does both. It uses **PICTURE ON** to store an image in **PICTURE\$**, and then the string is written to disk:

```
'record image
PICTURE ON
SHOWPEN
FOR i=1 TO 100
  m=MOUSE(0)
  x=MOUSE(1): y=MOUSE(2)
  LINE(x,y)-(x+10,y+10),,b
  LINE(490-x,253-y)-(490-x+10,253-y+10),,b
  LINE(x,253-y)-(x+10,253-y+10),,b
  LINE(490-x,y)-(490-x+10,y+10),,b
NEXT i
HIDEPEN
PICTURE OFF

'save string file
f$=FILES$(0,"save as string file:")
IF f$="" THEN END
OPEN f$ FOR OUTPUT AS 1
PRINT #1, PICTURE$
CLOSE 1
```

There are couple of points to notice about the program. The **FILES**\$(0) function is a very powerful way to interact with disk files. It brings up standard Macintosh dialog boxes to select files from the disk. **FILES**\$(0) prompts the user for a file name. **FILES**\$(1) lets the user select the name of an existing file. Also, the file format used in this program is simple sequential disk storage. It is not practical to store strings of varying length in a random access file.

Loading a string image back into memory from the disk is accomplished in much the same fashion, but with the order reversed. Enter the following:

```
'load string file
f$=FILES$(1)
IF f$="" THEN END
OPEN f$ FOR INPUT AS 1
image$=INPUT$(LOF(1),1)
CLOSE 1
```

```

'display image
FOR i=0 TO 200 STEP 40
  CLS
  PICTURE(i,0)-(490-i,253-i),image$
NEXT i

Stay: IF INKEY$="" THEN Stay

```

The INPUT\$(LOF(1),1) function reads the entire contents of the disk file into the string image\$.

ARRAY STORAGE ON DISK

Images loaded into arrays can also be transferred to disk. This next listing gets an image from the screen into an array and then stores it on disk:

```

'record image
DEFINT a-z
nframe=1
PENNORMAL
FOR i=1 TO 100
  m=MOUSE(0)
  x=MOUSE(1): y=MOUSE(2)
  LINE(x,y)-(x+10,y+10),,b
  LINE(490-x,253-y)-(490-x+10,253-y+10),,b
  LINE(x,253-y)-(x+10,253-y+10),,b
  LINE(490-x,y)-(490-x+10,y+10),,b
NEXT i
n=2+(((253-1)+1)*INT(((490-1)+16)/16))
DIM array(n)
GET (1,1)-(490,253),array

'save as array file
f$=FILES$(0,"Save as array file:")
IF f$="" THEN END
OPEN f$ FOR OUTPUT AS 1
PRINT #1,n;nframe;
FOR i=0 TO n
  PRINT #1, array(i);
NEXT i
CLOSE 1

```

This program stores the size of the array and the number of frames (nframe). You could also store the coordinates of the upper-left corner of the GET rectangle, which makes it possible to recreate the image in its original position.

Storing images in array format is very convenient, but there is a

price—an image stored in array format may require as much as three times the disk space that it would require in string format. A utility program that converts between the two formats is provided at the end of the chapter.

The next listing shows you how to load the image back into memory and display it on the screen:

```
'load array file
DEFINT a-z
InDiskA:
f$=FILES$(1)
IF f$="" THEN END
OPEN f$ FOR INPUT AS #1
INPUT #1,n,nframe
DIM array(n)
FOR i=0 TO n
    INPUT #1, array(i)
NEXT i
CLOSE #1

'display image
FOR i=0 TO 200 STEP 40
    CLS
    PUT ((x+1),y)-(490-i,253-i),array,PSET
NEXT i
Stay: IF INKEY$="" THEN Stay
```

The advantage of storing your images on disk is that they are there for instant recall, which eliminates having to slip in and out of BASIC to load images from other programs.

Sending Images to a Printer

If you're familiar with the Macintosh, you've probably used its built-in screen dump feature. When you press SHIFT-COMMAND-4, the Imagewriter prints a copy of the screen. Actually, there are two variations within this. If the CAPS LOCK key is up, only the current output window is printed. If the CAPS LOCK key is down, the entire screen is printed.

PRINTING LARGER IMAGES

The screen dump feature works fine for printing the output from most BASIC programs, but the Mac screen is clearly limited in size. What if you want to print something as large as an entire sheet of paper?

Fortunately, you can do this, because BASIC's limits on its statements are greater than the boundaries of the screen. If you direct a large image to the screen, all points that fall outside the screen boundaries are ignored. But if that same image is redirected to an output device with a larger drawing area (a printer), BASIC can take advantage of the larger dimensions.

The two statements you will use to redirect the output are `OPEN "lpt1:" FOR OUTPUT AS #1` and `WINDOW OUTPUT #1`. The `OPEN` statement opens the printer as a device, just like opening the Clipboard or a disk file. The device name `lpt1:` is reserved for the printer. The channel associated with this device is set to 1. The `WINDOW OUTPUT #1` statement sets up a direct link between the output window and channel 1, which represents the printer. In this way, anything "drawn" to the window will be routed to the printer instead. Because the printer is much larger than the screen, you can create a drawing window as large as the printer allows. On a standard Imagewriter, that size is 640 pixels (0-639) wide by 752 (0-751) pixels high.

The next step is to draw something on the window. The image is not displayed on the screen, but it is assembled in memory. When the window is closed, the image is converted to dot patterns that the printer understands and prints.

```
DEFINT a-z: DEFSGN t
WINDOW 1,"graphics",(0,0)-(639,751)
OPEN "lpt1:" FOR OUTPUT AS #1
WINDOW OUTPUT #1
LINE (0,0)-(639,751),,b
FOR t=0 TO 6.28 STEP .01
  x1=320+200*COS(t)*COS(4*SIN(2*t))
  y1=150+80*SIN(t)*COS(4*SIN(2*t))
  x2=320+200*COS(t)*COS(2*t)
  y2=600+80*SIN(t)*COS(2*t)
  LINE (x1,y1)-(x2,y2)
NEXT t
CLOSE #1
```

```
Stay: IF INKEY$="" THEN Stay
```

The `WINDOW 1` statement opens a window large enough for all the graphics statements issued. Figure 6-3 shows the results.

Now you can explore printing graphics in a larger format than the screen. Since Microsoft BASIC allows pixel coordinates from -32768 to 32767, you should be able to use output devices of any size.

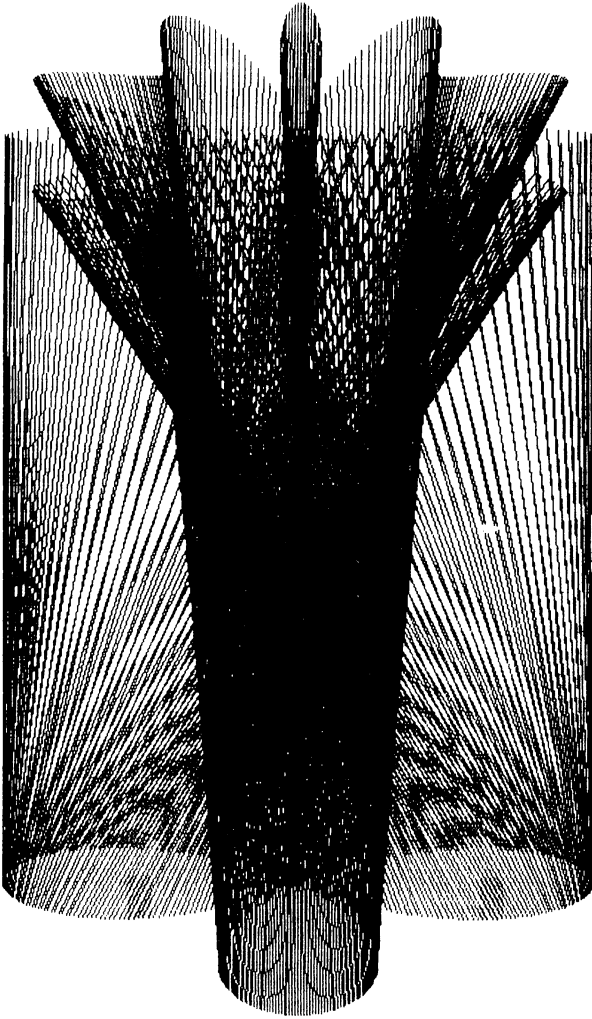


Figure 6-3.
Full-page printout

ADJUSTING FOR PRINTER DISTORTION

There is another subtle difference between using SHIFT-COMMAND-4 to copy the screen to the printer and using WINDOW OUTPUT # to draw to the printer. The screen dump utility controls the printer so that the printout reflects the 1-to-1 aspect ratio of the screen. But the WINDOW OUTPUT # statement uses the default horizontal spacing of the Imagewriter, which is 80 dots per inch, compared to the fixed 72 dots per inch of the vertical spacing. Thus, dots on the printout are closer together horizontally than they are on the screen, which makes circles look like ovals.

You can use two approaches to adjust for this distortion. The first is to change all your drawing commands to stretch out the horizontal coordinates by a factor of 80/72 (10/9), which makes the printout have a 1-to-1 aspect ratio. A second approach is to draw on-screen without any adjustments, using the PICTURE ON statement to record the graphics commands. Then use the PICTURE statement to redraw the image to the printer with the coordinates expanded by 10/9. This gets the job done with only one adjustment. The next listing shows the second approach:

```

DEFINT a-z: DEFSTR t
WINDOW 2,"graphics",(0,0)-(639,751)
SHOWPEN
PICTURE ON
LINE (100,110)-(300,220),,b
LINE (120,120)-(160,160),,b
CIRCLE(140,190),20
PICTURE OFF
HIDEPEN
OPEN "lpt1:" FOR OUTPUT AS #1
WINDOW OUTPUT #1
PICTURE (0,0),PICTURE$
MOVETO 170,148: PRINT"square"
MOVETO 170,196: PRINT"circle"
PICTURE (0,140)-(639*10/9,140+751),PICTURE$
MOVETO 190,288: PRINT"adjusted square"
MOVETO 190,336: PRINT"adjusted circle"
CLOSE #1: WINDOW CLOSE 2
Stay:IF INKEY$="" THEN Stay

```

Note that for the program to run correctly the first time, the WINDOW statement must appear before the image is recorded.

The second PICTURE statement uses the full range of the paper, (0,0)-(639,751), with two changes. First, 140 is added to the two y

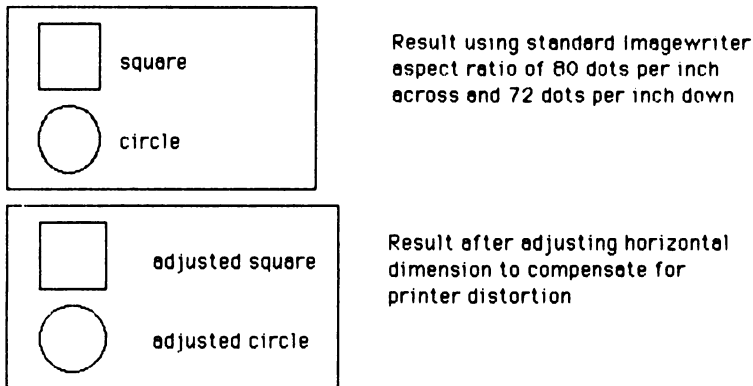


Figure 6-4.
Adjusting for printer distortion

coordinates to shift the figure down the page. Second, the maximum x value (639) is multiplied by 10/9 to stretch the images. Figure 6-4 shows the difference.

Applications and Ideas

There are several ways to manipulate images. This section will describe a few methods.

DIGITIZING IMAGES

One of the easiest ways to collect images to use in BASIC programs is with the assistance of a *digitizer*. A digitizer transfers complex images to a digitized format that a computer can use. There are several such units available for use with the Macintosh. A variety of methods can be used to capture the image, including using the printer as an input device, using a fixed-lens camera, using no lens at all, and using signals from a video camera or recorder.

The digitized sample shown in Figure 6-5 was created with Mac-Vision by Koala Technologies Corporation (3100 Patrick Henry Drive, Santa Clara, CA 95052-8100). The image was captured by a video



Figure 6-5.
A digitized image

camera and sent to the MacVision hardware unit. This magic box digitizes the image and displays it on the screen, where it can be copied into the Clipboard and pasted into MacPaint or loaded directly into BASIC. The required software installs easily as a desk accessory (in the Apple menu) and requires only about 6K of disk space.

The great advantage of digitized images is that they eliminate the need to develop graphics from scratch. The camera pulls the image into the computer; you can touch it up with FatBits if you want. Then it can be transferred to BASIC and manipulated as desired.

USING MACPAINT

MacPaint's FatBits feature is a natural for editing graphics images. Images created by applications programs, or even by BASIC, can be transferred in two ways: via the omnipresent Clipboard, or via the screen-dump-to-disk feature using SHIFT-COMMAND-3. Remember, if the CAPS LOCK key is up, SHIFT-COMMAND-3 copies the current window contents to a disk file. If the CAPS LOCK key is down, it dumps the entire screen to a disk file. The Mac will number these files sequentially as screen0, screen1, and so on. The best part of all is that files generated this way are totally compatible with MacPaint.

The screen dump feature is quick and easy, but it does have limitations. Screen images can easily use 20K of disk space apiece, so you must monitor disk space and keep close track of which image is stored in which screen file.

How can you transfer a MacPaint image back to BASIC? The standard method is again the Clipboard. Another way is to convert MacPaint files into a format that can be read by a BASIC program. A program called Paint Mover (MacinSoft, P.O. Box 27583, San Diego, CA 92128) can do this and more.

Paint Mover has several advantages over the Clipboard. It can convert an entire MacPaint document, whereas with the Clipboard, the image size is limited to the size of the MacPaint window. Paint Mover allows you to create images that fill an entire BASIC screen, something you cannot do with the Clipboard. Paint Mover can also take several MacPaint files and merge them into a single BASIC-compatible file. The images can be loaded into an array and displayed with PUT. Thus, several images can be readily available, subject to memory limitations. If you find yourself doing a lot of cutting and pasting between MacPaint and BASIC, Paint Mover would be a big time-saver.

What's more, the PUT statement is the most suitable for creating animation, since it permits the many display options necessary for preserving the background and masking display areas. Files created by Paint Mover are fully compatible with the PUT statement's data requirements. Using this package is probably the easiest way to create a large number of BASIC-compatible images for animation.

TRANSFER MODULES

Many transfer programs were introduced in this chapter. The next listing pulls them all together in one program so that you can see how they work together:

```
'initialize variables
RANDOMIZE TIMER
DEFINT a-z: DEFSTR t
flag=0
nframe=1

'set up menus
MENU 7,0,1,"String Transfer"
MENU 7,1,1,"Load from Clip"
MENU 7,2,1,"Save to Clip"
MENU 7,3,1,"Load from disk"
```

```

MENU 7,4,1,"Save to disk"
MENU 7,5,1,"Load from screen"
MENU 7,6,1,"Send to screen"
MENU 7,7,1,"Send to printer"
MENU 7,8,1,"Stop"
MENU 8,0,1,"Array Transfer"
MENU 8,1,1,"Load from disk"
MENU 8,2,1,"Save to disk"
MENU 8,3,1,"Load from screen"
MENU 8,4,1,"Send to screen"
MENU 8,5,1,"Send to printer"
MENU 8,6,1,"Stop"

```

DoMenu:

```

MENU
msg$="Select menu item"
GOSUB Message
scan: m0=MENU(0)
IF m0<7 OR m0>8 THEN scan
ON m0-6 GOTO menu7, menu8

```

menu7:

```

ON MENU(1) GOSUB InClip, OutClip, InDisk, OutDisk, InScreen, OutScreen, OutPrinter, halt
GOTO DoMenu

```

menu8:

```

ON MENU(1) GOSUB InDiskA, OutDiskA, InScreenA, OutScreenA, OutPrinterA, halt
GOTO DoMenu

```

InClip:

```

OPEN "CLIP:PICTURE" FOR INPUT AS 1
i$=INPUT$(LOF(1),1)
IF i$="" THEN msg$=" >>> Clipboard is empty <<<": GOSUB Message
IF i$<>"" THEN msg$=">>> Clipboard recorded in string <<<": GOSUB Message: image$=i$
CLOSE 1
RETURN

```

OutClip:

```

IF image$="" THEN msg$=" >>> String is empty <<<": GOSUB Message: RETURN
OPEN "CLIP:PICTURE" FOR OUTPUT AS 1
PRINT #1, image$
CLOSE 1
msg$="String recorded in Clipboard": GOSUB Message
RETURN

```

InDisk:

```

f$=FILES$(1)
IF f$="" THEN RETURN
OPEN f$ FOR INPUT AS 1
msg$="Loading string file"
delay=1
GOSUB Message
image$=INPUT$(LOF(1),1)
CLOSE 1
RETURN

```

OutDisk:

```

f$=FILES$(0,"save as string file:")
IF f$="" THEN RETURN
OPEN f$ FOR OUTPUT AS 1
PRINT #1, image$
CLOSE 1
RETURN

```

InScreen:

```

CLS
WINDOW 2,,(100,100)-(400,200),2
BUTTON 2,1,"from array",(80,20)-(200,40),2
BUTTON 3,1,"from drawing",(80,60)-(250,80),2

```

Dial:

```

IF DIALOG(0)>1 THEN Dial
WINDOW 1
PICTURE ON
SHOWPEN
IF DIALOG(1)=2 THEN GOSUB OutScreenA: IF flage=0 THEN PICTURE OFF: RETURN
IF DIALOG(1)=3 THEN GOSUB Draw
PICTURE OFF
image$=PICTURE$
msg$="Picture recorded"
GOSUB Message
RETURN

```

OutScreen:

```

IF image$<>"" THEN PICTURE (0,30),image$: RETURN
msg$=">>> Load string first <<<"
GOSUB Message
RETURN

```

OutPrinter:

```

IF image$="" THEN msg$=">>> Load string first <<<": GOSUB Message: RETURN
WINDOW 2,"printing",(0,0)-(639,751)

```

```

OPEN 1pt1:" FOR OUTPUT AS #1
WINDOW OUTPUT #1
PICTURE ,image$
CLOSE #1
WINDOW 1
RETURN

```

```

InDiskA:
f$=FILES$(1)
IF f$="" THEN RETURN
msg$= ">>> Loading array file <<<"
delay=1
GOSUB Message
OPEN f$ FOR INPUT AS 1
IF flag=1 THEN ERASE a
INPUT #1,n,nframe
DIM a(n): flag=1
FOR i=0 TO n
    INPUT #1, a(i)
NEXT i
CLOSE 1
RETURN

```

```

OutDiskA:
IF flag=0 THEN msg$=">>> Load array first <<<": GOSUB Message: RETURN
f$=FILES$(0,"Save as array file:")
IF f$="" THEN RETURN
msg$= "Saving array file"
delay=1
GOSUB Message
OPEN f$ FOR OUTPUT AS 1
PRINT #1,n,nframe;
FOR i=0 TO n
    PRINT #1, a(i);
NEXT i
CLOSE 1
RETURN

```

```

InScreenA:
CLS
WINDOW 2,,(100,100)-(400,200),2
BUTTON 2,1,"from string",(40,20)-(200,40),2
BUTTON 3,1,"from drawing",(40,60)-(250,80),2

```

```

Dial2:
IF DIALOG(0)>1 THEN Dial2
WINDOW 1
IF DIALOG(1)=2 THEN GOSUB OutScreen: IF image$="" THEN RETURN
IF DIALOG(1)=3 THEN GOSUB Draw

```

```

pat%(1)= -21931
pat%(2)= -21931
pat%(3)= -21931
pat%(4)= -21931

InSloop:
msg$= "select rectangle with mouse"
GOSUB Message
WHILE MOUSE(0)=0
  ENDO
h1=MOUSE(3)
y1=MOUSE(4)
h2=h1
y2=y1
PENPOT DRAFTER(pat%(1))
PENMODE 10
IF MOUSE(0)>0 THEN InSloop
WHILE MOUSE(0)<0
  r(1)=y1
  r(3)=y2
  IF y2<y1 THEN r(1)=y2: r(3)=y1
  r(2)=h1
  r(4)=h2
  IF h2<h1 THEN r(2)=h2: r(4)=h1
  FRAMERECT DRAFTER(r(1))
  h3=h2
  y3=y2
  WHILE (h3=h2 AND y3=y2)
    z=MOUSE(0)
    h3=MOUSE(1)
    y3=MOUSE(2)
  ENDO
  FRAMERECT DRAFTER(r(1))
  h2=h3
  y2=y3
ENDW
h1=r(2)
y1=r(1)
h2=r(4)-1
y2=r(3)-1

GetIt:
PENNORMAL
FRAMERECT DRAFTER(r(1))
IF flag=1 THEN ERASE a
n=2+((y2-y1)+1)*INT(((h2-h1)+16)/16)
DIM a(n)
flag=1
GET(h1,y1)-(h2,y2),a
msg$= ">>> rectangle selected <<<"
GOSUB Message

```

```
CLS
RETURN
```

```
OutScreenA:
  IF flag=0 THEN msg$=">>> Load array first <<<":
    GOSUB Message: RETURN
  CLS
  PUT (x1,y1)-(x2,y2),a,PSET
RETURN
```

```
OutPrinterA:
  IF flag=0 THEN msg$=">>> Load array first <<<": GOSUB Message: RETURN
  WINDOW 2,"printing",(0,0)-(639,751)
  OPEN "pt1:" FOR OUTPUT AS #1
  WINDOW OUTPUT #1
  PUT (0,0),a
  CLOSE #1
  WINDOW 1
RETURN
```

```
Halt:
  MENU RESET
```

```
END
```

```
Draw:
  CLS
  FOR t=0 TO 6.28 STEP .01
    x1=245+200*COS(t)*COS(4*SIN(2*t))
    y1=126+80*SIN(t)*COS(4*SIN(2*t))
    x2=245+200*COS(t)*COS(2*t)
    y2=126+80*SIN(t)*COS(2*t)
    LINE(x1,y1)-(x2,y2)
  NEXT t
RETURN
```

```
Message:
  BUTTON 1,1,msg$(0,0)-(500,20)
  FOR d=1 TO delay*2000
    NEXT d
    delay=5
RETURN
```

STRING/ARRAY CONVERSION

You can use the complete module listing to build a customized transfer utility program quickly. For example, this next program converts directly between the string file format and the array file format:

```

DEFINT a-z
nframe=1
flaga=0
abort=0
pat$(1)= -21931
pat$(2)= -21931
pat$(3)= -21931
pat$(4)= -21931

Start:
CLS
WINDOW 2,,(100,100)-(400,200),2
BUTTON 2,1,"String to array",(80,20)-(200,40),2
BUTTON 3,1,"Array to string",(80,60)-(250,80),2

```

```

Dial:
IF DIALOG(0)<>1 THEN Dial
WINDOW 1
IF DIALOG(1)=2 THEN GOSUB StringToArray
IF DIALOG(1)=3 THEN GOSUB ArrayToString
GOTO Start

```

```

StringToArray:
GOSUB InDisk
IF abort=1 THEN abort=0: RETURN
GOSUB OutScreen
GOSUB InScreenA
GOSUB OutDiskA
RETURN

```

```

ArrayToString:
GOSUB InDiskA
IF abort=1 THEN abort=0: RETURN
GOSUB InScreen
GOSUB OutDisk
RETURN

```

```

InDisk:
f$=FILES$(1)
IF f$="" THEN abort=1: RETURN
OPEN f$ FOR INPUT AS 1
image$=INPUT$(LOF(1),1)
CLOSE 1
RETURN

```

```

OutDisk:
f$=FILES$(0,"save as string file.")
IF f$="" THEN abort=1: RETURN
OPEN f$ FOR OUTPUT AS 1

```

```

PRINT #1, image$
CLOSE 1
RETURN

```

```

InScreen:
  PICTURE ON
  SHOWPEN
  GOSUB OutScreenA
  PICTURE OFF
  IF flag=0 THEN RETURN
  image$=PICTURE$
RETURN

```

```

OutScreen:
  IF image$="" THEN RETURN
  PICTURE (0,30),image$
RETURN

```

```

InDiskA:
  f$=FILES$(1)
  IF f$="" THEN abort=1
  OPEN f$ FOR INPUT AS 1
  IF flag=1 THEN ERASE a
  INPUT #1,n,nframe
  DIM a(n)
  flag=1
  FOR i=0 TO n
    INPUT #1, a(i)
  NEXT i
  CLOSE 1
RETURN

```

```

OutDiskA:
  IF flag=0 THEN RETURN
  f$=FILES$(0,"Save as array file:")
  IF f$="" THEN abort=1: RETURN
  OPEN f$ FOR OUTPUT AS 1
  PRINT #1,n,nframe;
  FOR i=0 TO n
    PRINT #1, a(i);
  NEXT i
  CLOSE 1
RETURN

```

```

InScreenA
  IF image$="" THEN RETURN
  PICTURE (0,30),image$
  msg$= "Use the mouse to select the section of the design you want."
  BUTTON 1,1,msg$(0,0)-(500,20)
  GOSUB Selectrect

```



```

PENNORMAL
FRAMERECT DRAFTER(r(1))
n=1+((r(3)-r(1))+1)*INT(((r(4)-r(2))+16)/16)
IF flaga=1 THEN ERASE a
DIM a(n)
flaga=1
GET(r(2),r(1))-(r(4)-1,r(3)-1),a
CLS
BUTTON CLOSE 1
RETURN

```

```

OutScreenA:
IF flaga=0 THEN RETURN
CLS
PUT(0,30),a,PSET
RETURN

```

```

Selectrect:
WHILE MOUSE(0)=0
WEND
x1=MOUSE(3)
y1=MOUSE(4)
x2=x1
y2=y1
PENPAT DRAFTER(pat%(1))
PENMODE 10
IF MOUSE(0)>=0 THEN Selectrect
WHILE MOUSE(0)<0
r(1)=y1
r(3)=y2
IF y2<y1 THEN SWAP r(1),r(3)
r(2)=x1
r(4)=x2
IF x2<x1 THEN SWAP r(2),r(4)
FRAMERECT DRAFTER(r(1))
x3=x2
y3=y2
WHILE (x3=x2 AND y3=y2)
z=MOUSE(0)
x3=MOUSE(1)
y3=MOUSE(2)
WEND
FRAMERECT DRAFTER(r(1))
x2=x3
y2=y3
WEND
RETURN

```

This program was built by modifying each of the modules needed from the previous program.

MEMO PAD

We've all seen the ubiquitous "From the desk of" memo pads. This program combines text and pictures to give them a new twist:

```

PRINT "Copy the Mac picture from the Scrapbook"
INPUT "Hit return to continue", x
INPUT "Enter your name >", myname$
CLS
OPEN "CLIP-PICTURE" FOR INPUT AS #1
i$=INPUT$(LOF(1),1)
CLOSE #1
WINDOW 2,"memo",(0,0)-(639,751)
PICTURE ON
SHOWPEN
TEXTMODE 1
TEXTFONT 1
TEXTSIZE 18
PICTURE (70,5),i$
MOVETO 10,20
PRINT "From the"
MOVETO 215,100
PRINT "of "; myname$
HIDEPEN
PICTURE OFF
memo$ = PICTURE$
OPEN "lpt1:" FOR OUTPUT AS #1
WINDOW OUTPUT #1
PICTURE ,memo$
CLOSE #1

```

To create the image, you must copy the Macintosh picture that Apple provides in the System Master disk's Scrapbook into the Clipboard. Next, enter your name, and the program will print your notepad logo, as shown in Figure 6-6.

If you haven't noticed, the program does something we haven't mentioned yet: it includes PICTURE statements within a picture-creation routine to make a new picture. And, if you like, you can write the new picture to the Clipboard, add it to the Scrapbook, and use it in your MacWrite documents.

Summary

The Macintosh has many tools that make it easy to manipulate and transfer images in ways never before possible on a microcomputer. This chapter discussed two formats for storing images: string and array. String storage consists of a string of QuickDraw com-



Figure 6-6.
Creating your notepad logo

mands that can be used to recreate the image. Array storage maintains a bit-by-bit representation of the image.

Using one or both of these storage formats, you can transfer images among memory, screen, disk, and Clipboard, and even out to the printer. Which format you select depends on a variety of factors, including the speed at which you wish to recreate the image, memory and disk storage requirements, and compatibility with other programs.

The tools developed in this chapter pave the way for more complex graphics abilities. In Chapter 7, these tools will be used to explore the possibilities of animating images with Microsoft BASIC.

7

Animation Techniques

By now you know how to draw images on the Macintosh screen and save them on a disk. In this chapter, you will learn how to apply these skills to one of the most captivating aspects of computer graphics: animation.

Literally, to *animate* means to give life to (*à la* Doctor Frankenstein). In practice, to animate means to give the appearance of motion where there actually is none. This illusion is created in animated cartoons by the rapid presentation of thousands of drawings in a sequence, with each drawing slightly different from its predecessor. The human mind then interprets the sequence of images as continuous motion.

Conventional animators must sketch each of these drawings by hand. They have ways of saving time and effort, but with pencil,

eraser, and paper, the process is still tedious. However, the graphics capabilities of the Macintosh open up new doors for you, the aspiring animator. Using Microsoft BASIC graphics statements, you can write programs that draw animation sequences for you. You can also use the image transfer techniques (developed in Chapter 6) to help you animate hand-drawn (actually, mouse-drawn) images.

In this chapter, you will learn some of the basic techniques of computer animation. You begin by learning to move a fixed object around the screen, using two versions of the PUT statement. You then learn to display a sequence of still images, called *frames*, in rapid succession to create believable animation. The chapter concludes with utility programs that help you create and store frames for animation.

The subject of animation can easily fill several volumes. The goal of this chapter is to cover enough of the basics to get you started in the right direction. For a more thorough coverage of the subject, read *Macintosh Game Animation*, by Ron Person (Osborne/McGraw-Hill, 1985).

Single-Frame Motion

A good way to start learning about animation is to move a single object around the screen. In the previous chapter, you learned how to store objects in both string and array formats and how to display them anywhere on the screen. This chapter uses both methods, but we will start with array storage, since it is more interesting.

Recall that the PUT statement transfers a bit image from an array to the screen using one of the Microsoft BASIC action verbs—PSET, PRESET, AND, OR, and XOR. These control the interaction between the stored image and the contents of the current screen. The two most useful action verbs for animation are PSET and XOR. If you do not specify an action verb in the PUT statement, BASIC will assume XOR. PSET means that the current screen contents are totally obliterated by the incoming stored image. XOR stands for exclusive OR; its effects are summarized in Table 7-1.

The XOR action described in Table 7-1 determines the color of each pixel when a stored image is displayed on the screen with PUT. The pixel will be black if either the incoming pixel or the current screen pixel is black, but not if both are black. The pixel will be white if the screen and the stored image pixels are both black or both white. In short, if both pixels are the same color, the resulting pixel will be white; otherwise, it will be black.

Table 7-1.
XOR Action

Screen	Stored Image	Result
black	black	white
black	white	black
white	black	black
white	white	white

MOVING AN OBJECT

The first program uses the PSET action verb to move a ball (a filled circle) around the screen:

```

DEFINT a-z
x=100: y=100: radius=25: dx=4: dy=4
x1=x-radius: x2=x+radius: y1=y-radius: y2=y+radius
size=2+(((y2-y1)+1)*INT(((x2-x1)+16)/16))
DIM block(size)
PRINT "Press any key to pause and resume the program"
PRINT "Press Mouse button to stop"
FOR i=1 TO 20000: NEXT i
CLS
CIRCLE(x,y),radius
r%(0)=y-radius+1: r%(1)=x-radius+1
r%(2)=y+radius: r%(3)=x+radius
p%(0)=-26266:p%(1)=17025:p%(2)=-32446:p%(3)=26265
FILLOVAL VARPTR(r%(0)),VARPTR(p%(0))
GET (x1,y1)-(x2,y2),block

```

```

Animation:
PUT (x-radius,y-radius),block,PSET
x=x+dx: y=y+dy
IF x<radius THEN x=radius: dx=-dx
IF x>465 THEN x=465: dx=-dx
IF y<radius THEN y=radius: dy=-dy
IF y>275 THEN y=275: dy=-dy
IF MOUSE(0)<0 THEN END
IF INKEY$="" THEN animation
stay: IF INKEY$="" THEN stay
GOTO Animation

```

Press any key to pause and resume the action. Click the mouse pointer in the output window to stop the program.

Figure 7-1 shows what can happen if you are not careful with PSET motion. The ball leaves a trail as it moves, but this effect is due

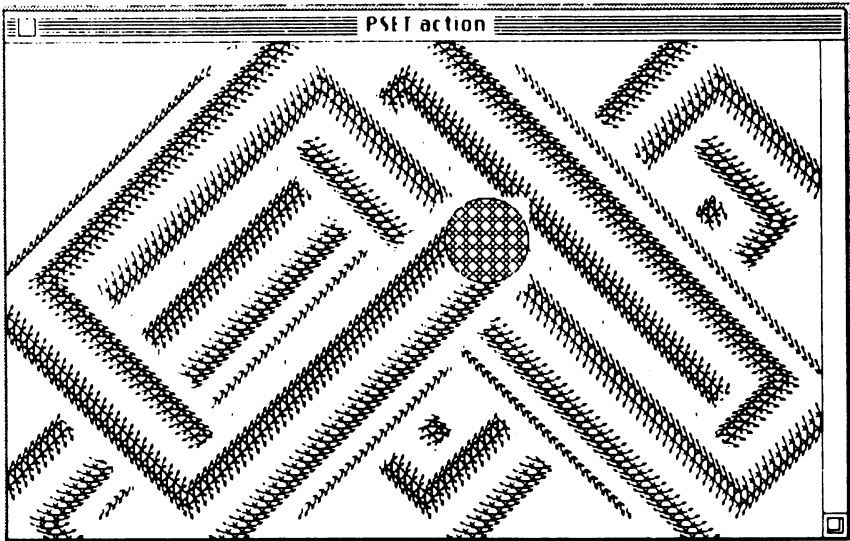


Figure 7-1.
PSET action leaving a trail

only to a lack of planning: the GET rectangle is the same size as the circle. When the stored image is moved to a new location with PUT, the next version doesn't completely cover the old one, so it leaves a trail. The golden rule for PSET animation is to leave a border between the object and the GET rectangle that is at least as wide as the number of pixels the object is to be moved.

The problem can be corrected by increasing the boundary of the rectangle by four pixels in all directions, since the ball is moving four pixels at a time. Change these lines:

```
x1=x-radius-dx: x2=x+radius+dx
y1=y-radius-dy: y2=y+radius+dy
```

The four-pixel border erases any trail, no matter which direction the ball travels. The same technique works just as well on a black background, as long as you use black for the border pattern. Make this change:

```
' ...
CLS
LINE(0,0)-(490,342),,bf
CIRCLE(x,y),radius
' ...
```


ANIMATING ON A BACKGROUND PATTERN

Moving an object over a background requires even more planning. Delete the LINE statement and make the following changes:

```
'...
GET (x1,y1)-(x2,y2),block
r%(0)=0: r%(1)=0: r%(2)=342: r%(3)=491
p%(0)=225: p%(1)=15653: p%(2)=9533: p%(3)=15869
FILLRECT VARPTR(r%(0)),VARPTR(p%(0))
```

Animation:

```
'...
```

This program leaves a trail of background pattern. One way to prevent this is to save the background before each PUT statement and then replace the background after each PUT statement. Fortunately, BASIC offers an easier solution: use XOR motion instead of PSET motion. Make these changes:

```
Animation:
nx=x-radius: ny=y-radius
PUT (nx,ny),block,XOR
'...
IF MOUSE(0)<0 THEN END
PUT (nx,ny),block,XOR
IF INKEY$="" THEN Animation
```

Using XOR twice retains the background. The first XOR merges with the background, turning all shared points white (see Table 7-1). The second XOR fills these shared points back in and then deletes the others. Thus, the PUT image is erased and the background is restored.

One disadvantage of XOR animation is that the object tends to flicker. You can reduce this effect by adding a small time delay (for example, FOR i=1 TO 100: NEXT i) between the two PUTs to the same location.

Multiple-Frame Animation

The next level of animation is to change the shape of a stationary object. There are several ways to accomplish this. One method is to display the entire object and then use simple graphics statements like PSET and LINE to modify portions of it. This gives the effect of motion.

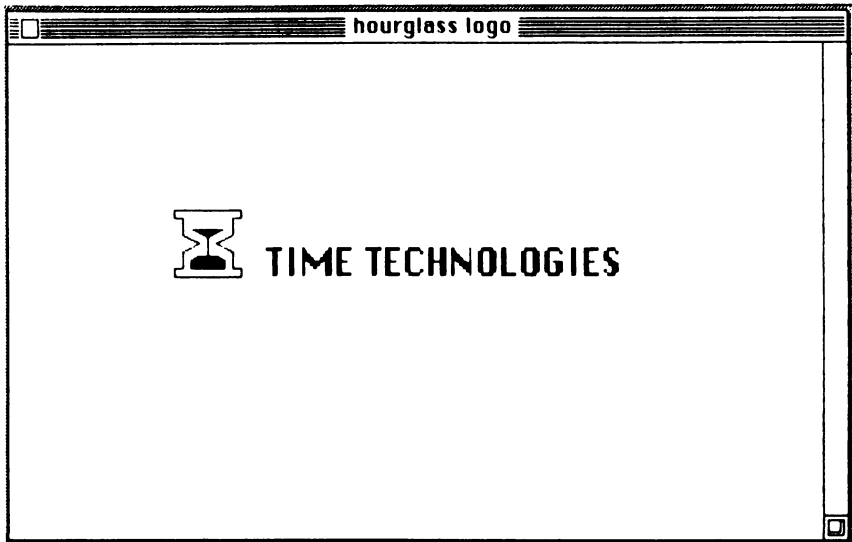


Figure 7-2.
Animation hourglass logo

CONTINUOUS MOTION

The animated logo shown in Figure 7-2 is a modification of a logo used in Chapter 3. This time, the sand sifts from the top chamber to the bottom one, just as it does in an hourglass. The effect of motion occurs when the black area in the top chamber is erased and black lines are added to the bottom chamber at the same time. The vertical column of sand is erased from the top down with PRESET to finish the cycle. Since the figure is quite small, BASIC is fast enough to make the motion appear almost continuous. In fact, the program contains some delay loops to slow down the motion. Here is the program:

```

DEFINT a-z
DIM glass(42), sand(16), eraser(3)
FOR i=0 TO 42: READ glass(i): NEXT i
DATA 86,10,10,10,10,10,100,100,105,100,105,105,115,105
DATA 120,115,125,105,135,105,135,100,140,100,140
DATA 140,135,140,135,135,125,135,120,125,115,135,105
DATA 135,105,140,100,140,100,100
FOR i=0 TO 16: READ sand(i): NEXT i
DATA 34,105,110,117,130,105,110,112,110,117,120,112
DATA 130,105,130,105,110
FOR i=0 TO 3: READ eraser(i): NEXT i
DATA 102,110,105,130

```

```

FRAMEPOLY VARPTR(glass(0))
PAINTPOLY VARPTR(sand(0))
LINE (120,117)-(120,135)
TEXTFONT 0: TEXTSIZE 22
MOVETO 155,140
PRINT "TIME TECHNOLOGIES"
X=110
Y=135
X1= 130

FOR i=1 TO 14
  FOR j=1 TO 1000: NEXT j
  ERASERECT VARPTR(eraser(0))
  eraser(2) = eraser(2) + 1
  IF i < 11 THEN LINE(X,Y)-(X1,Y)
  Y=Y-1
  IF i > 5 THEN X= X + 2: X1 = X1 - 2
NEXT i
FOR i=117 TO 124
  FOR j=1 TO 100: NEXT j
  PRESET(120,i)
NEXT i
FOR i=1 TO 5000: NEXT i
RUN

```

Another approach is to select a portion of the object and replace it with a controlled sequence of images. For example, the next program displays a woman's face and then winks one eye (Figure 7-3). The wink is produced by using PUT with a carefully selected sequence of frames that show the eyelid in different positions. Figure 7-4 shows the sequence.

Notice that the images are small and manageable. Not only do they take up a minimal amount of room in memory, but they can also be displayed by the PUT statement fast enough to produce believable animation.

Also notice that the change from frame to frame is subtle. When the frames are displayed in order, the motion appears smooth and continuous. In creating your own sequences, be sure to include enough frames so that the viewer's mind will not need to fill in too many gaps.

STORING IMAGES

The eye frames in Figure 7-4 were created with MacPaint and then transferred to BASIC via the Clipboard. This process seems easy until you stop to think about where you are going to store the images. The ideal scenario would be to display the entire sequence on the

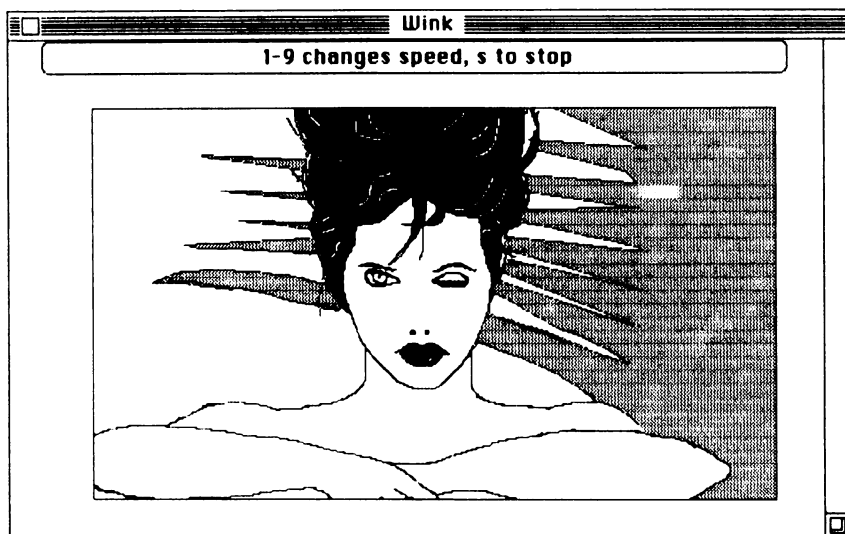


Figure 7-3.
Winking face

screen and then somehow grab the images one at a time into equal-sized arrays. The trick here is knowing how to select uniformly sized frames around each image so that the eye will be in the same position in each rectangle. One solution would be to draw borders around each image in MacPaint and write a custom program to GET the images into a two-dimensional array.

The approach used for the eye sequence shown in the figure was to write a utility program that allowed the operator to use the mouse to select a border rectangle and position that rectangle uniformly around each image. The images were then stored in a disk file where they could be loaded into an array and animated with PUT. (The program is listed as MacPaint Transfer; it appears at the end of the chapter.)



Figure 7-4.
Wink sequence

You'll find that preparing and storing the frames is the major battle in frame sequence animation. Animating the images once they are stored on disk is comparatively easy.

Here is the complete program listing:

```
'initialize variables
DEFINT a-z

LoadFace:
OPEN "MS-Basic graphics:face.pic(1)" FOR INPUT AS 1
INPUT #1,n,nframe
DIM aw(n)
  msg$="loading Face.Pic": GOSUB Message
  FOR i=0 TO n
    INPUT #1,aw(i)
  NEXT i
BUTTON CLOSE 1
CLOSE 1

LoadEyes:
OPEN "MS-Basic graphics:eyes.pic(10)" FOR INPUT AS 1
INPUT #1,n,nframe
DIM a(n,nframe)
FOR f=1 TO nframe
  msg$="loading eye frame "+STR$(f): GOSUB Message
  FOR i=0 TO n
    INPUT #1,a(i,f)
  NEXT i
NEXT f
BUTTON CLOSE 1
CLOSE 1

Animate:
msg$="1-9 changes speed, s to stop": GOSUB Message
PUT (50,40),aw,PSET
LINE (50,40)-(462,276),,b
k=1
AniLoop:
FOR i=1 TO nframe
  PUT (252,133),a(0,i),PSET
  IF i=1 THEN FOR x=1 TO 9000: NEXT x
  i$=INKEY$
  IF i$="s" OR i$="S" THEN END
  v=VAL(i$): IF v>0 THEN k=10-v
  FOR j=1 TO k*100: NEXT j
NEXT i
GOTO AniLoop
CLS: BUTTON CLOSE 1

Quit:
END
```

Message:

```
BUTTON 1,1,msg$(20,0)-(470,20)
FOR d=1 TO delay*2000: NEXT d
delay=5
RETURN
```

The first two sections of the program load in the picture of the woman and the eye frames from two separate files. Both files use the same format. The first number in the file is the number of cells necessary to store each frame. The second number is the number of frames stored in the file. Use these numbers to dimension an array large enough to hold the images. Be sure to include the appropriate disk and file names in the OPEN statements.

The animation section of the program displays the woman. It then displays the eye-frame sequence, controlled by time delays. The program uses the INKEY\$ function to scan for keyboard activity. Keys 1 through 9 change the speed of the wink. The 1 key is slowest, 9 is fastest, and S stops the program.

ROTATING OBJECTS

A more daring feat is to create a separate frame for each view of the entire object instead of changing just a small portion of it. The problem with storing the entire object in separate frames is twofold. First, it requires lots of memory. Users with 128K Macintoshes will quickly encounter the "Out of Memory" message. Second, it takes more time to cover the entire screen with dots than it does to cover a small section. If the image is too large, the eye can detect that the image is put on the screen in sections.

Figure 7-5 shows six frames representing different views of the world as it rotates on its axis. Draw the frames with MacPaint and then transfer and animate them with the MacPaint Transfer program. Watching these frames displayed in sequence definitely gives one the impression of a rotating globe, but the animation is jerky. Can you guess why?

The problem is not with the PUT statement. The frames are still small enough that the images appear "instantly" on the screen. Size is part of the problem. As image size increases, the number of frames must increase also. The animator must provide more steps to maintain the illusion of continuous motion. The challenge here is to select the minimum number of frames that will produce believable animation for a given frame size.

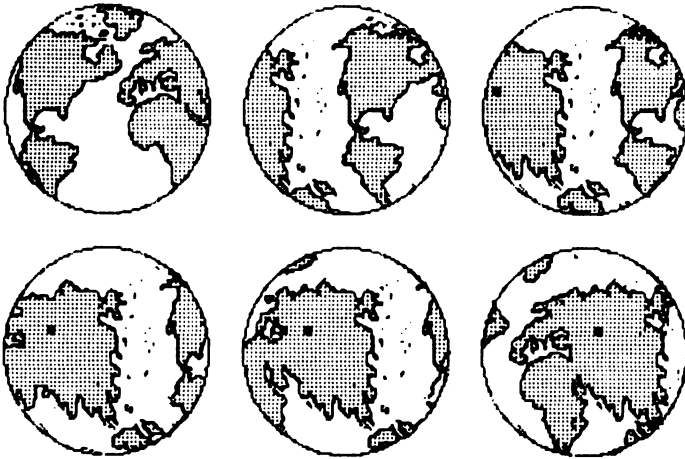


Figure 7-5.
Six views of the world

DIGITIZING FRAMES

Using MacPaint to create animation frames is very convenient, as long as the size of the images is reasonable. For complex images, you may want to use a digitizer to capture individual frames. Figure 7-6 illustrates what a digitizer can do.

Koala's MacVision unit captured each of these images from a videotape signal (see Chapter 6 for more information). It then transferred them into the Macintosh through the terminal port. Using the freeze-frame feature of a videotape unit, you can select individual frames. This sequence shows approximately every third video frame.

As you can see, the frames in Figure 7-6 could stand considerable touching up before they might be ready to animate in BASIC. Even so, using digitized images can give you a tremendous head start toward creating large, complex images for animation.

GENERATING FRAMES IN BASIC

Creating animation frames in MacPaint is not an exact science. Even with the MacPaint Transfer utility program, centering images uniformly within the frame boundaries requires either careful planning

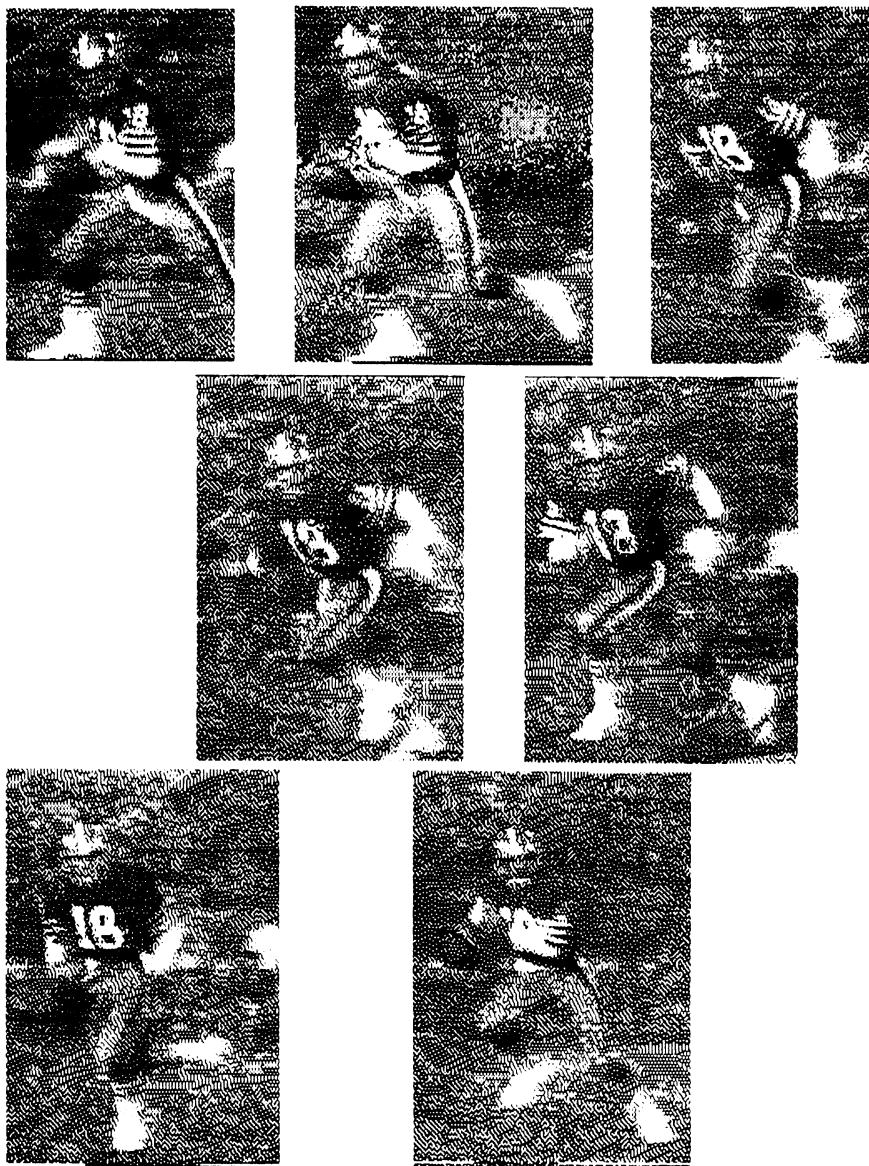


Figure 7-6.
Digitized football player

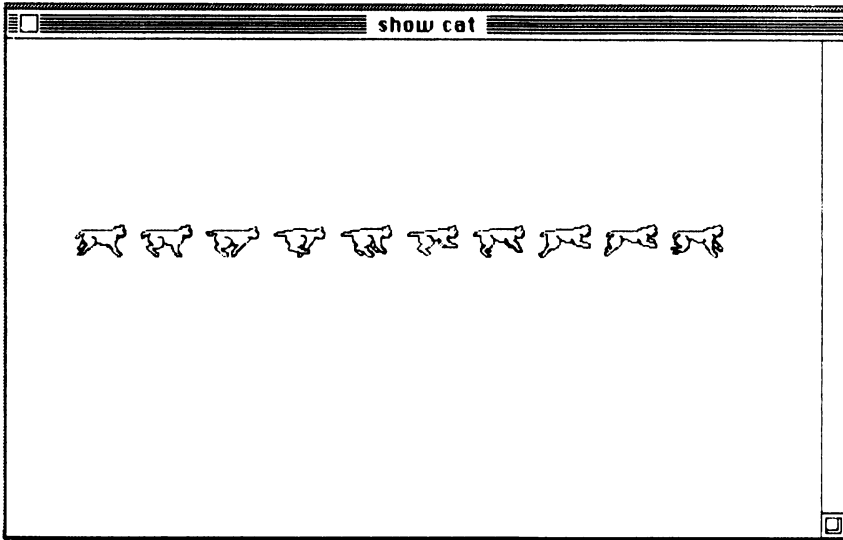


Figure 7-7.
Cat in motion

or a very steady hand. For more control, you can create images pixel by pixel using a BASIC program.

The sequence in Figure 7-7 was created with just such a program. The program is called Animator; it is listed in the “Animation Utility Program” section later in the chapter. Animator creates frames that are 32 pixels square and saves them to disk.

The following program animates these frames:

```

DEFINT a-z

LoadFile:
f$=FILES$(1)
IF f$="" THEN STOP
PRINT"Loading file"
OPEN f$ FOR INPUT AS #1
INPUT #1,n,nframe
DIM a(n,nframe)
FOR frame=1 TO nframe
  FOR i=0 TO n
    INPUT #1,a(i,frame)
  NEXT i
NEXT frame
CLOSE #1
    
```

```
CLS
PRINT "1-9 changes speed, s TO STOP"
```

```
k=1
Animate:
FOR x=1 TO 491 STEP nframe
  FOR i=1 TO nframe
    PUT(x+i,201),a(0,i),PSET
    LINE(x+i-1,201)-(x+i-1,300),white
    i$=INKEY$
    IF i$="s" OR i$="S" THEN END
    v=VAL(i$): IF v>0 THEN k=10-v
    FOR j=1 TO k*100: NEXT j
  NEXT i
NEXT x
GOTO Animate
```

Note that the LINE statement erases any trail left by pixels in the far left column of each frame. You can change the PSET action in the PUT statement to XOR if you prefer. Just remember that you get a slight flicker with XOR motion.

Animation Utility Programs

The two utility programs in this section are designed to help you manage animation.

ANIMATOR UTILITY

The first program allows you to create your own animation frames and store them on disk for use with this (or another) program:

```
DEFINT a-z
n1=301: n2=332: y1=51: y2=82
n= 2+(((y2-y1)+1)*INT(((n2-n1)+16)/16))
frame=1: nframe=1: fsize=32
max=20: 'adjust to 6 for 128K Mac
DIM pat(fsize,fsize,max),pattern(3), a(n,max),temp(n)
FOR i=0 TO max
  a(0,i)=32: a(1,i)=32
NEXT i
```

```
Start:
WINDOW 2,"animator",(0,40)-(511,340),1
ON DIALOG GOSUB boxes: DIALOG ON
GOSUB DrawGrid
msg$="Editing frame ~: GOSUB Message
BUTTON 2,1,"LOAD FRAME",(47,250)-(157,280),1
```

```

BUTTON 3,1,"SAVE FRAME",(167,250)-(277,280),1
BUTTON 4,1,"LOAD FILE",(400,30)-(470,55),1
BUTTON 5,1,"SAVE FILE",(400,60)-(470,85),1
BUTTON 6,1,"ANIMATE",(400,90)-(470,115),1
BUTTON 7,1,"SHIFT >>>",(400,120)-(470,145),1
BUTTON 8,1,"QUIT",(400,150)-(470,175),1

```

CheckMouse:

```

IF MOUSE(0)=0 THEN flag=0: GOTO CheckMouse
H←MOUSE(1): Y←MOUSE(2): H3←MOUSE(3)
IF ABS(161-H)≥111 OR ABS(133-Y)≥111 THEN CheckMouse
C←INT(H/7)-6
R←INT(Y/7)-2
IF H≠H3 THEN flag=flag+1
IF flag=1 AND H≠H3 THEN color=pat(C,R,0)
IF color≠pat(C,R,0) THEN CheckMouse
pat(C,R,0)=1-pat(C,R,0)
IF pat(C,R,0)=1 THEN LINE(7*C+44,7*R+16)-(7*C+47,7*R+19),bf: PSET(300+C,50+R)
IF pat(C,R,0)=0 THEN LINE(7*C+44,7*R+16)-(7*C+47,7*R+19),30,bf: PSET(300+C,50+R)
GOTO CheckMouse

```

Boxes:

```

d←DIALOG(0)
IF d>1 THEN RETURN
IF DIALOG(1)=2 THEN GOSUB LoadFrame
IF DIALOG(1)=3 THEN SaveFrame
IF DIALOG(1)=4 THEN GOSUB LoadFile
IF DIALOG(1)=5 THEN GOSUB SaveFile
IF DIALOG(1)=6 THEN GOSUB Animate
IF DIALOG(1)=7 THEN GOSUB ShiftRight
IF DIALOG(1)=8 THEN GOSUB Quit
msg$="editing frame": GOSUB Message
RETURN

```

Quit:

```

WINDOW 1
END

```

LoadFrame:

```

DIALOG OFF: BUTTON 2,0
EDIT FIELD 1,STR$(frame),(310,100)-(330,115)
Inpt1: IF DIALOG(0)≥6 THEN Inpt1
frame←VAL(EDIT$(1))
IF frame>max OR frame<0 THEN BEEP: frame=1: GOTO LoadFrame
EDIT FIELD CLOSE 1
GOSUB GetFrame
DIALOG ON: BUTTON 2,1
RETURN

```

SaveFrame:

```

DIALOG OFF: BUTTON 3,0
EDIT FIELD 1,STR$(frame),(310,100)-(330,115)
msg$="Press Return": GOSUB Message
Inpt2: IF DIALOG(0)>6 THEN Inpt2
frame=VAL(EDIT$(1))
IF frame>max OR frame<0 THEN DEEP: frame=1: GOTO SaveFrame
IF frame>nframe THEN nframe=frame
EDIT FIELD CLOSE 1
msg$="saving frame "+STR$(frame): GOSUB Message
GET(h1,y1)-(h2,y2),a(0,frame)
FOR r=1 TO fsize
  FOR c=1 TO fsize
    pat(c,r,frame)=pat(c,r,0)
  NEXT c
NEXT r
msg$="editing frame "
GOSUB Message
DIALOG ON: BUTTON 3,1
RETURN

```

SaveFile:

```

f$=FILES$(0,"Save as array file:")
IF f$="" THEN RETURN
GOSUB DrawGrid
FOR r=1 TO 17
  FOR c=1 TO fsize
    IF pat(c,r,0)=1 THEN LINE(7*c+44,7*r+16)-(7*c+47,7*r+19),,bf
  NEXT c
NEXT r
msg$="Saving file "+f$: GOSUB Message
OPEN f$ FOR OUTPUT AS 1
PRINT #1, n;nframe;
FOR j=1 TO nframe
  FOR i=0 TO n
    PRINT #1, a(i,j);
  NEXT i
NEXT j
CLOSE #1
msg$="Editing frame "+STR$(frame): GOSUB Message
RETURN

```

LoadFile:

```

f$=FILES$(1)
IF f$="" THEN RETURN
BUTTON 4,0
LINE(49,21)-(273,245),30,bf
GOSUB DrawGrid
OPEN f$ FOR INPUT AS #1
INPUT #1,n,nframe

```

```

ERASE a,pot
DIM a(n,max),pot(fsize,fsize,max)
FOR frame=1 TO nframe
  msg$="Loading frame "+STR$(frame): GOSUB Message
  FOR i=0 TO n
    INPUT #1,a(i,frame)
  NEXT i
  PUT (301,51),a(0,frame),PSET
  FOR r=1 TO fsize
    FOR c=1 TO fsize
      IF POINT(300+c,50+r)=33 THEN pot(c,r,frame)=1
    NEXT c
  NEXT r
NEXT frame
frame=1
CLOSE #1
PUT (301,51),a(0,0),PSET
BUTTON 4,1
RETURN

```

```

Animate:
  msg$="1-9 changes speed, s to stop": GOSUB Message
  k=1
  AniLoop:
    FOR i=1 TO nframe
      PUT(301,201),a(0,i),PSET
      i$=INKEY$
      IF i$="s" OR i$="S" THEN Anileave
      v=VAL(i$): IF v>0 THEN k=10-v
      FOR j=1 TO k*100: NEXT j
    NEXT i
    GOTO AniLoop
  Anileave:
    LINE(301,201)-(332,232),30,bf
  RETURN

```

```

Message:
  BUTTON 1,1,msg$(50,0)-(270,20)
  RETURN

```

```

DrawGrid:
  FOR h=49 TO 273 STEP 7
    LINE(h,21)-(h,245)
    LINE(49,h-28)-(273,h-28)
  NEXT h
  RETURN

```

```

GetFrame:
  msg$="Loading frame "+STR$(frame): GOSUB Message
  PUT (301,51),a(0,frame),PSET

```

```

FOR r=1 TO fsize
  MOVE TO 280,7*r+20: TEXTMODE 2: PRINT "<";
  FOR c=1 TO fsize
    IF pat(c,r,frame)=1 THEN LINE(7*c+44,7*r+16)-(7*c+47,7*r+19),,bf
    IF pat(c,r,frame)=0 THEN LINE(7*c+44,7*r+16)-(7*c+47,7*r+19),30,bf
    pat(c,r,0)=pat(c,r,frame)
  NEXT c
  MOVE TO 280,7*r+20: PRINT "<"; TEXTMODE 0
NEXT r
RETURN

```

ShiftRight:

```

msg$="shifting frame right": GOSUB Message
BUTTON 7,0
FOR c=31 TO 1 STEP -1
  FOR r=1 TO fsize
    SWAP pat(c,r,0),pat(c+1,r,0)
  NEXT r
NEXT c
FOR r=1 TO fsize
  MOVE TO 280,7*r+20: TEXTMODE 2: PRINT "<";
  FOR c=1 TO fsize
    IF pat(c,r,0)=1 THEN LINE(7*c+44,7*r+16)-(7*c+47,7*r+19),,bf
    IF pat(c,r,0)=0 THEN LINE(7*c+44,7*r+16)-(7*c+47,7*r+19),30,bf
  NEXT c
  MOVE TO 280,7*r+20: PRINT "<"; TEXTMODE 0
NEXT r
GET(332,51)-(332,82),temp
GET(301,51)-(331,82),a(0,0)
PUT(302,51),a(0,0),PSET
PUT(301,51),temp,PSET
BUTTON 7,1
RETURN

```

Two of the design considerations were to keep the image size small and to store the frames in files that would easily load into array format. The program uses a grid size of 32×32 , which is small enough to be manageable for very smooth animation. Even so, the Animator program only allows six frames in the 128K Mac. Be sure to set variable Max to 6 or less if you use the program on a 128K machine. On 512K (Fat) Macintoshes, you can set Max to 20 or higher.

Once stored on disk, the frames can be loaded into memory and animated. If you intend to use PSET animation, be sure to leave

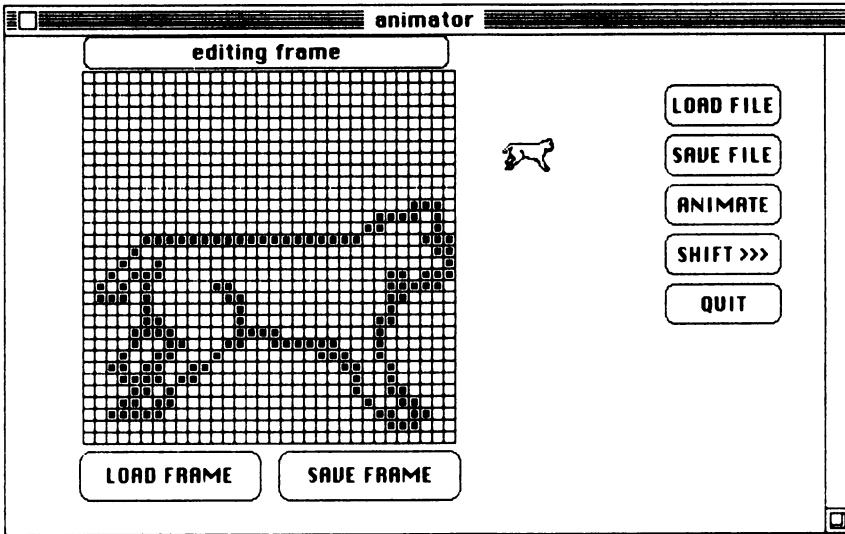


Figure 7-8.
Animator screen

enough blank columns so that the image doesn't leave a trail or erase the trail as you did in the previous listing.

When you run the Animator program, be sure to watch the button above the grid for messages (see Figure 7-8). It alerts you to the status of the program and prompts you for action when necessary.

Sketch frames by clicking and dragging the mouse. Save each frame to memory with **SAVE FRAME** when it is complete. If you want to use a frame already drawn as the basis for a new frame, load it with **LOAD FRAME**. This takes time, so be patient. There is also a button for shifting the edit frame right one pixel, which is useful for animating objects that rotate.

You can animate at any time with **ANIMATE**. Press keys 1 through 9 to vary speed and S to stop animation.

When you are satisfied with the sequence of frames, save it to disk with **SAVE FILE**. With **LOAD FILE** you can load frame sequences into the program for further editing or load them into other BASIC programs.

If you really want to get serious about animation, try the Animation Toolkit 1 from Ann Arbor Softworks, Incorporated,

308 1/2 South State Street, Ann Arbor, Michigan 48104, (313) 996-3838. This program is written in assembly language, so it is very fast. It has many editing features and stores frames very compactly. It can handle 140 32×32 frames on a 128K Mac and up to 3,000 frames on a 512K Mac. Ann Arbor Softworks also provides a utility that converts frame sequences into Macintosh fonts that can be used in BASIC or other programs.

MACPAINT TRANSFER UTILITY

This is a program for transferring MacPaint images into array format. In particular, the program allows you to select a frame size and use it to collect several images from a MacPaint file. The images are then stored in a single file. The steps in this process are as follows:

1. Load the screen from the Clipboard into a string variable.
2. Display the image in the BASIC output window.
3. Use the mouse to select a rectangular frame size.
4. Place the rectangle around each image and store it on disk.

The program also lets you load a file into memory and animate it. You can copy an image out of the Scrapbook to exercise the program. Users of 128K Macs should keep in mind that the size and number of frames are limited.

When you run the program, be prepared with the file name and the number of frames you wish to use. Here is the program:

```
'initialize variables
DEFINT a-z: flag=0
pat%(1)=-21931:pat%(2)=-21931
pat%(3)=-21931:pat%(4)=-21931

Choose:
LINE(160,40)-(350,165),,b
BUTTON 2,1,"Select frames",(180,60)-(330,85),1
BUTTON 3,1,"Animate frames",(180,90)-(330,115),1
BUTTON 4,1,"Quit",(180,120)-(330,145),1
Dial:
d0=DILOG(0): IF d0>1 THEN Dial
d=DILOG(1)
IF d<2 OR d>4 THEN Dial
BUTTON CLOSE 2
BUTTON CLOSE 3
BUTTON CLOSE 4
CLS
```



```
ON d-1 GOSUB SelectFrames, AnimateFrames,Quit
GOTO Choose
```

SelectFrames:

```
INPUT "Enter number of frames to be stored";nframe
f$=FILES$(0,"Save frames in file:")
IF f$="" THEN STOP
OPEN f$ FOR OUTPUT AS 1
```

Inclip:

```
CLS
OPEN "CLIP:PICTURE" FOR INPUT AS 2
i$=INPUT$(LOF(2),2)
IF i$="" THEN msg$=">>> Clipboard is empty <<<: GOSUB Message: RETURN
IF i$<>"" THEN msg$=">>> Clipboard recorded in string <<<: GOSUB Message: image$=i$
CLOSE 2
OutScreen:
IF image$<>"" THEN PICTURE (0,30),image$: GOTO SelectRectangle
msg$=">>> Load string first <<<"
GOSUB Message
```

SelectRectangle:

```
InsLoop:
msg$="select rectangle with mouse"
GOSUB Message
WHILE MOUSE(0)=0: WEND
x1=MOUSE(3): y1=MOUSE(4)
x2=x1: y2=y1
PENPAT DARPTR(pat%(1))
PENMODE 10
IF MOUSE(0)>=0 THEN InsLoop
WHILE MOUSE(0)<0
  r(1)=y1: r(3)=y2
  IF y2<y1 THEN r(1)=y2: r(3)=y1
  r(2)=x1: r(4)=x2
  IF x2<x1 THEN r(2)=x2: r(4)=x1
  FRAMERECT DARPTR(r(1))
  x3=x2: y3=y2
  WHILE (x3=x2 AND y3=y2)
    z=MOUSE(0)
    x3=MOUSE(1): y3=MOUSE(2)
  WEND
  FRAMERECT DARPTR(r(1))
  x2=x3: y2=y3
WEND
x1=r(2): y1=r(1): x2=r(4)-1: y2=r(3)-1
h=y2-y1: w=x2-x1
```

```

VerifyRectangle:
msg$="Rectangle okay (y/n)?": GOSUB Message
in: i$=INKEY$
IF i$="" THEN in
IF i$="n" THEN SelectRectangle
n=2+((y2-y1)+1)*INT(((x2-x1)+16)/16)
IF flag=1 THEN ERASE a
DIM a(n): flag=1

FOR f=1 TO nframe
  PositionRectangle:
  msg$="Select frame "+STR$(f): GOSUB Message
  WHILE MOUSE(0)>=0
    x1=MOUSE(1): y1=MOUSE(2)
    x2=x1: y2=y1
    r(1)=y2-h: r(2)=x2: r(3)=y2: r(4)=x2+w
    FRAMERECT DRAWPTA(r(1))
    FOR i=1 TO 500: NEXT i
    FRAMERECT DRAWPTA(r(1))
  WEND
  msg$="frame selected": GOSUB Message
  'GetIt
  x1=r(2): y1=r(1): x2=r(4)-1: y2=r(3)-1
  GET(x1,y1)-(x2,y2),a
  'OutDiskA
  msg$=" Saving frame "+STR$(f): GOSUB message
  IF f=1 THEN PRINT #1,n;nframe;
  FOR i=0 TO n
    PRINT #1, a(i);
  NEXT i
NEXT f
CLOSE 1
CLS: BUTTON CLOSE 1
RETURN

```

```

Message:
  BUTTON 1,1,msg$(20,0)-(470,20)
  FOR d=1 TO delay*2000: NEXT d
  delay=5
RETURN

```

```

AnimateFrames:
f$=FILES$(1)
IF f$="" THEN STOP
OPEN f$ FOR INPUT AS 1
INPUT #1,n,nframe
IF flag=1 THEN ERASE a
DIM a(n,nframe): flag=1
FOR f=1 TO nframe
  msg$="Loading frame "+STR$(f): GOSUB Message
  FOR i=0 TO n

```

```

    INPUT #1,a(i,f)
  NEXT i
NEXT f
BUTTON CLOSE 1
CLOSE 1

'Animate
msg$="1-9 changes speed, s to stop": GOSUB Message
k=1
AniLoop:
FOR i=1 TO nframe
  PUT (50,50),a(0,i),PSET
  i$=INKEY$
  IF i$="s" OR i$="S" THEN AniLeave
  v=VAL(i$): IF v>0 THEN k=10-v
  FOR j=1 TO k*100: NEXT j
NEXT i
GOTO AniLoop
AniLeave:
CLS: BUTTON CLOSE 1
RETURN

Quit:
END

```

The first two numbers saved in the file represent the size of the array elements (*n*, determined by the GET rectangle) and the number of frames (*nframe*). These two numbers make it easy to load the file. The rest of the file contains data for the frames. This is the same file format as that used in the Animator program. You may want to modify the Animator utility so that it accepts variable-sized arrays created by the MacPaint Transfer program.

Summary

You have now had a taste of how the Macintosh can help you animate images. This chapter showed you how to animate single-frame, program-drawn objects. It also demonstrated techniques for combining multiple pre-drawn frames into animation sequences.

The next chapter will introduce you to techniques you can use to manipulate Macintosh graphics. You will see how to move, rotate, and stretch objects in two and three dimensions. You will even learn some computer-aided design techniques.

8

Manipulating Displays And Viewing Objects

In this chapter you will see how to manipulate points, lines, polygons, and even complex screen images with BASIC. The programs include routines for three types of transformations: translations (moving the object around the screen), rotations (changing the orientation of the object), and scaling (changing the size of the object). The routines will be extended to include viewing and manipulating three-dimensional objects. A miniature computer-aided design (CAD) program, originally written by Rob Dickerson of Microsoft and modified for use in this book, will conclude the chapter.

Back in Chapter 2, we mentioned the Macintosh's huge coordinate system, ranging from $-32,768$ to $+32,767$. The Mac screen displays only a portion of this graphics area. We will now take a closer look at the relationship between this coordinate system and the video display. This background material will give you a firm conceptual basis for manipulating objects on the screen.

Caution: This chapter delves briefly into some mathematics, par-

ticularly trigonometry. Don't let this put you off. Try reading the material slowly to give yourself time to absorb the ideas. You can make good use of the programs without getting too involved with the math.

Transformations

The Macintosh is loaded with tools that help programmers change the appearance of objects on the screen. The term that describes these changes is *transformation*. We will describe three transformations: moving an object from one location to another (translation), rotating an object about a fixed point (rotation), and changing the size of an object (scaling). The discussion will begin with transformations of objects that contain only two dimensions: length and width. Two-dimensional objects like circles and squares can be displayed on a flat surface. Later in the chapter, the techniques developed for two-dimensional objects will be extended to cover representations of objects containing three dimensions: length, width, and depth. Three-dimensional objects include those you see around you every day — cars, people, and even computers.

Two-Dimensional Transformations

The simplest object in a two-dimensional plane is a point, which is located by its horizontal (x) and vertical (y) distances from the origin of the coordinate system. For our purposes, the coordinate system refers to the numbers used by the BASIC output window (look back at Chapter 2, Figure 2-3). Recall that this is not the same as the Cartesian coordinate system, which is typically used in mathematics (refer to Figure 2-12 for a refresher).

For each of the two-dimensional transformations discussed here, we list equations that apply the transformation to a point (x,y) to get a new point (XNEW,YNEW). Then we will use those equations in a sample program.

TRANSLATION

To *translate* (move) a point from one location on the screen to another, simply add the horizontal and vertical distance values to its x and y coordinates, respectively. Figure 8-1 shows the effect of adding the values h (horizontal distance) and k (vertical distance) to the point (x,y).

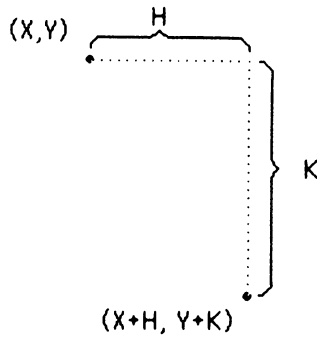


Figure 8-1.
Translating a point

If you want to move an entire object, add the translation values h and k to each of its points. Why use a point-by-point approach when the GET and PUT statements can do the job neatly and efficiently? Because you can rapidly and easily translate objects that are described by a few control points by translating only those points. Consider, for example, a polygon. You need only translate the corner points and then connect them by using BASIC's LINE statement. Figure 8-2 shows an object relocated in this way.

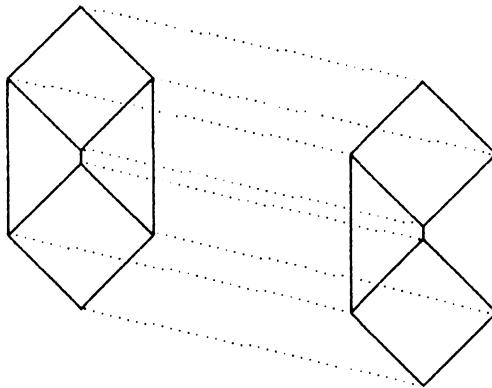


Figure 8-2.
Translating a polygon

SCALING

You *scale* an object by multiplying the coordinates of each of its points by a constant. Scaling also moves the object in the plane. If the scaling factor is less than one (but greater than zero), it compresses the object and moves it closer to the origin. If the factor is greater than one, scaling enlarges the object and moves it away from the origin. If you scale both coordinates by the same factor, the object maintains its original proportions. By using different scaling factors for horizontal and vertical coordinates, you can distort the object. To reflect an object, use a negative scaling factor. For example, scaling horizontal coordinates by -1 reflects (flips) the object horizontally. Figure 8-3 illustrates the effects of scaling on a polygon.

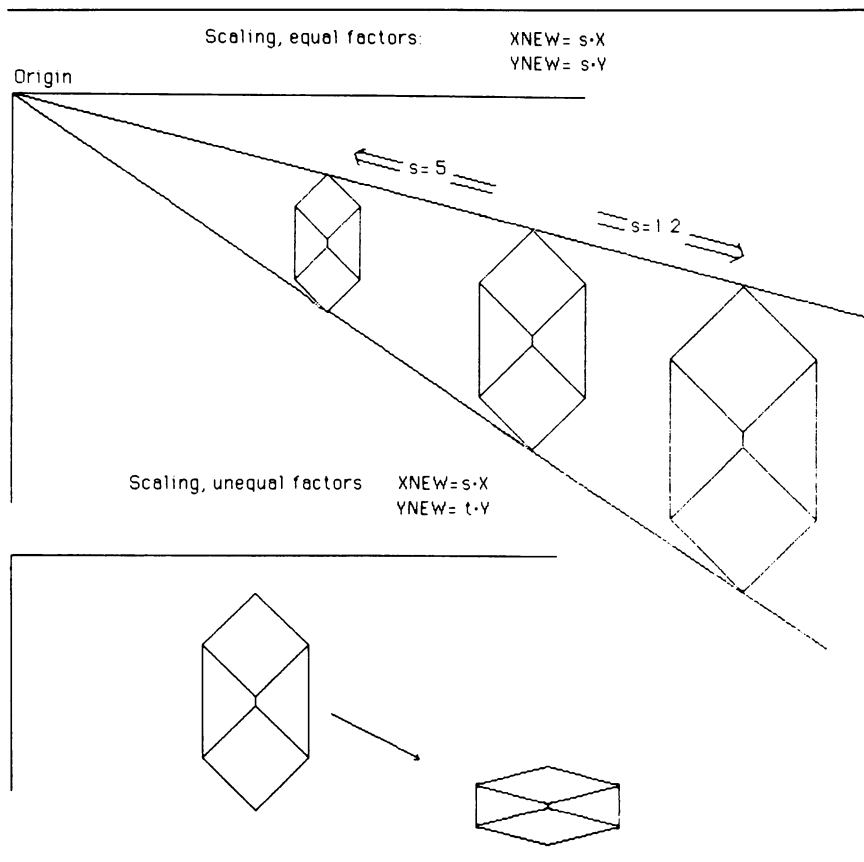
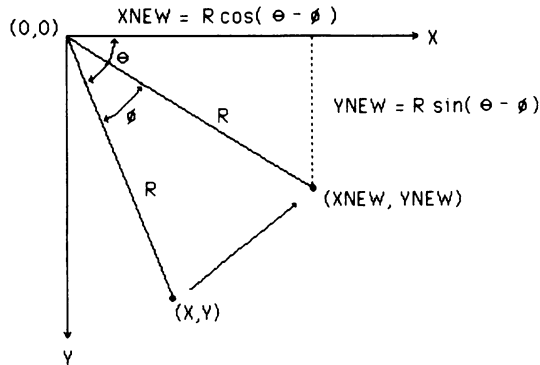


Figure 8-3.
Scaling polygons

ROTATION

You can use BASIC to rotate objects. Once again, the technique involves rotating a single point. To rotate a complete object, apply the rotation equations to each of its points.

Figure 8-4 illustrates how to rotate a point (x,y) through the angle ϕ to $(XNEW,YNEW)$ and derives the rotation equations. The equations use the trigonometric sine and cosine functions. These functions are essential for dealing with angles, and you don't have to be a trigonometry expert to use the equations.



$$\begin{aligned}
 XNEW &= R \cos(\theta - \phi) \\
 &= R (\cos\theta \cos\phi + \sin\theta \sin\phi) \\
 &= (R \cos\theta) \cos\phi + (R \sin\theta) \sin\phi \\
 &= X \cos\phi + Y \sin\phi
 \end{aligned}$$

$$\begin{aligned}
 YNEW &= R \sin(\theta - \phi) \\
 &= R (\sin\theta \cos\phi - \cos\theta \sin\phi) \\
 &= (R \sin\theta) \cos\phi - (R \cos\theta) \sin\phi \\
 &= Y \cos\phi - X \sin\phi
 \end{aligned}$$

$$\begin{aligned}
 XNEW &= X \cos\phi + Y \sin\phi \\
 YNEW &= Y \cos\phi - X \sin\phi
 \end{aligned}$$

Figure 8-4.
Equations of two-dimensional rotation

USING TRANSFORMATION EQUATIONS IN BASIC

Let's try out the three transformations on a simple straight line. The plan is to make a subprogram for each of the transformation equations. Once the subprograms are in place, you can call each subprogram as necessary to transform both endpoints of the line and then use the LINE statement to redraw the rest of the line. In this program, you will first translate the line, then rotate it, and finally scale it, as shown in Figure 8-5.

TRANSFORMATION SUBPROGRAMS

The subprograms you need are listed below:

```
Subprograms:
SUB Translate(x,y,h,k) STATIC
  x=x+h: y=y+k
END SUB
SUB Rotate(x,y,a) STATIC
  c=COS(a): s=SIN(a)
  xt=x*c+y*s
  yt=y*c-x*s
  x=xt: y=yt
END SUB
SUB Scale(x,y,s,t) STATIC
  x=s*x
  y=t*y
END SUB
```

Before using these program segments, there are several things you should notice. First, these are not subroutines; they are subprograms. A subprogram is a new kind of program module described in the Advanced Topics chapter of the *Microsoft BASIC Manual*. Review the chapter now if you are unfamiliar with subprograms.

Subprograms are especially convenient in program segments that will be reused in several different programs. The variables listed in the SUB subprogram statement are used as temporary names for the corresponding variables in the calling statement. They serve the same purpose as the list of names in a function definition (DEF FN) statement.

Consider, for example, the statement CALL translate (xold,yold,xmove,ymove). The variables x, y, h, and k in the SUB translate

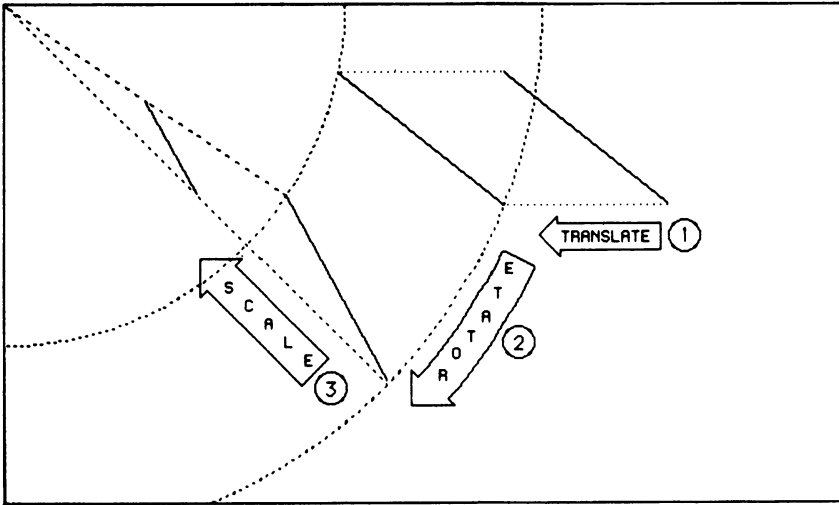


Figure 8-5.
Transforming a line

statement (see listing) are temporarily assigned as names for the variables `xold`, `yold`, `xmove`, and `ymove`, respectively. Thus, the statement `x=x+h` actually performs the operation `xold=xold+xmove`.

The rotation subprogram deserves special comment. The rotation angle, a , is expressed in an angular measure known as radians. Recall from Chapter 3 that BASIC's `SIN` and `COS` functions will only accept angles in radians, rather than in the more familiar degrees. The relationship between radians and degrees is

$$360 \text{ degrees} = 2 \times \pi \text{ radians where } \pi (\pi) \text{ is approximately } 3.1416$$

Positive angles result in counterclockwise rotation. Negative angles rotate points clockwise. Note also that the `COS` and `SIN` functions are relatively slow. To save time, the subprogram calculates `COS(a)` and `SIN(a)` once, and then reuses the calculated values.

The rotation program uses temporary variables `xt` and `yt` to avoid trouble. If it used `x` and `y` instead, the equation `y=y*c-x*s` would

use a modified x from the previous program line, rather than the original value passed to the subprogram. Using xt and yt as temporary variables avoids this problem.

Note: We have used the statement `DEFINT a-z` throughout this book. This declares all variables as integer types and speeds up the programs. When working with transformations, you will usually need the accuracy of single-precision variables. Thus, the programs in this chapter use the default (for Binary BASIC) single-precision type and define integer variables only as needed.

USING THE TRANSFORMATION SUBPROGRAMS

Insert these lines before your subprograms:

```
x1=300: y1=40
x2=400: y2=120
LINE(x1,y1)-(x2,y2)
h=-100: k=0
translate x1,y1,h,k
translate x2,y2,h,k
LINE(x1,y1)-(x2,y2)
a=-.4
rotate x1,y1,a
rotate x2,y2,a
LINE(x1,y1)-(x2,y2)
s=.5: t=.5
scale x1,y1,s,t
scale x2,y2,s,t
LINE(x1,y1)-(x2,y2)
Stay: IF INKEY$="" THEN Stay
END
```

The revised program assigns values to the endpoints of the line, (x1,y1) and (x2,y2), and then draws the line. Next it assigns values to h and k and calls the translate subprogram for each endpoint. Notice that the word `CALL` is optional, and not used in this program.

Next the program assigns the value -0.4 to a, rotates the endpoints, and draws the rotated line. This angle rotates the line clockwise by approximately 23 degrees. The program applies a scaling factor of 0.5 to the x and y coordinates of the endpoints of the last line. Then it draws the line. Press any key to stop the program.

TRANSFORMATIONS ABOUT A FIXED POINT

The scaling and rotation routines have a basic limitation. In their current form, both routines transform a point about the origin (look back at Figure 8-5). To rotate or scale an object about an arbitrary point (s,t), as in Figure 8-6, perform these three steps: first, translate

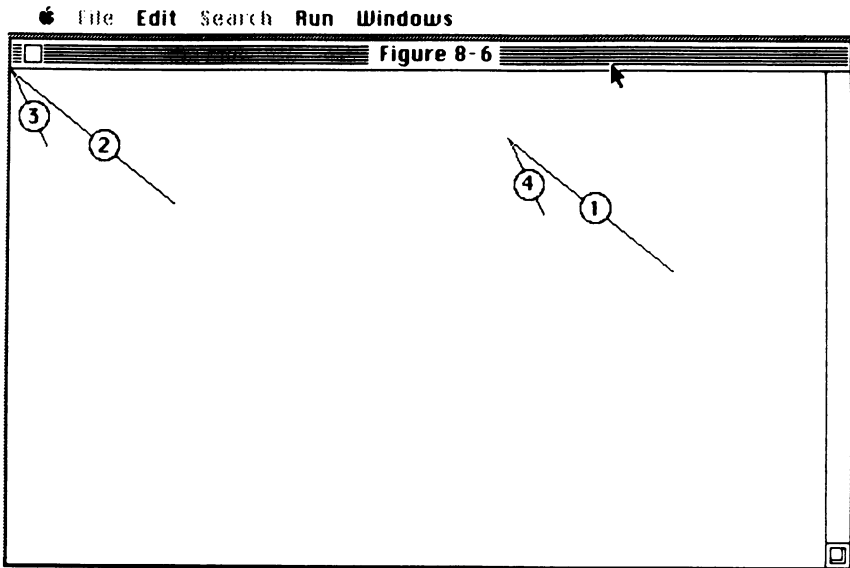


Figure 8-6.

The original line at (1) has been translated to the origin at (2); at (3) it has been rotated and scaled; (4) shows the line translated back

the object's points by $(-s, -t)$. Then rotate or scale the object. Finally, translate again by (s, t) . The next listing demonstrates this approach. It rotates and translates a line about one of its endpoints:

```
x1=300: y1=40
x2=400: y2=120
LINE(x1,y1)-(x2,y2)
h=-x1: k=-y1
Translate x1,y1,h,k
Translate x2,y2,h,k
LINE(x1,y1)-(x2,y2)
a=-.55
Rotate x2,y2,a
s=.5: t=.5
Scale x2,y2,s,t
LINE(x1,y1)-(x2,y2)
Translate x1,y1,-h,-k
Translate x2,y2,-h,-k
LINE(x1,y1)-(x2,y2)
Stay: IF INKEY$="" THEN Stay
END
```

Subprograms:

SUB Translate(x,y,h,k) STATIC

$x=x+h$: $y=y+k$

END SUB

SUB Rotate(x,y,a) STATIC

$c=\cos(a)$: $s=\sin(a)$

$xt=x*c+y*s$

$yt=y*c-x*s$

$x=xt$: $y=yt$

END SUB

SUB Scale(x,y,s,t) STATIC

$x=s*x$

$y=t*y$

END SUB

You can avoid these extra translations by adding a reference point to the subprograms. The following listing shows how to rotate an object about a fixed point (x0,y0):

SUB Rotate(x,y,x0,y0,a) STATIC

$c=\cos(a)$: $s=\sin(a)$

$xt=x0+(x-x0)*c+(y-y0)*s$

$yt=y0+(y-y0)*c-(x-x0)*s$

$x=xt$: $y=yt$

END SUB

In the next program, you will rotate the letter M about a point (x0,y0) like this:



The program stores the letter as a polygon. It updates each point in the polygon array with this rotation subprogram:

SetUp:

DEFINT h,k,p,x,y

$pi=3.1415926*$

$ai=-2*pi/25$

$a=ai$

DIM poly(14), polyt(14), pgone(14)

$pgone(0)=30$

FOR i=0 **TO** 14: **READ** poly(i): $polyt(i)=poly(i)$: **NEXT** i

DATA 30,46,36,104,104,100,40,50,40,70,70,50,100,100,100

$x0=70$: $y0=75$

FOR r=1 **TO** 5: **FOR** c=1 **TO** 2

READ path(r,c)

DATA 4,-4,8,-2,8,2,4,6,2,10

```

NEXT c,r
' *** draw stairs ***
sx=20: sy=103
FOR i=1 TO 10
  LINE(sx+26,sy)-(sx+26,sy+12)
  LINE(sx,sy)-(sx+26,sy)
  sx=sx+26: sy=sy+12
NEXT i

' *** draw A, C, and first M ***
LINE(370,220)-(400,170): LINE(400,170)-(430,220)
LINE(382,200)-(418,200): CIRCLE (460,195),25,,8,5.48
FRAMEPOLY VARPTR(poly(0))

Tumble:
FOR rep=1 TO 10
  FOR m=1 TO 5
    ca=COS(a): sa=SIN(a)
    FOR i=1 TO 14: pgone(i)=polyt(i): NEXT i
    Translate x0,y0,path(m,1),path(m,2)
    FOR i=1 TO 14 STEP 2
      Translate poly(i+1),poly(i),path(m,1),path(m,2)
      polyt(i)=poly(i): polyt(i+1)=poly(i+1)
      IF i>4 THEN Rotate polyt(i+1),polyt(i),x0,y0,ca,sa
    NEXT i
    PENPAT 380
    FRAMEPOLY VARPTR(pgone(0))
    PENPAT 492
    FRAMEPOLY VARPTR(polyt(0))
    a=a+ai
  NEXT m
NEXT rep
Stay: IF INKEY$="" THEN Stay
END

REM ** Subprograms **
SUB translate(x,y,h,k) STATIC
  x=x+h: y=y+k
END SUB
SUB rotate(x,y,x0,y0,c,s) STATIC
  xt=x0+(x-x0)*c+(y-y0)*s
  yt=y0+(y-y0)*c-(x-x0)*s
  x=xt: y=yt
END SUB

```

The program also translates the M before each rotation to make the visual effect more interesting. The path array contains the amounts by which the M is translated. Figure 8-7 shows the cumulative positions of the letter M as it tumbles down the stairs.

The letter is erased by storing the polygon in an auxiliary array, pgone. Just before it draws the next letter, the program erases the

old letter by calling FRAMEPOLY with the pen pattern set to 380 (white). It then changes the pen pattern back to black with PENPAT 492.

Notice that the program does not rotate the original polygon representation of the M, kept in the array poly. Instead, it makes a temporary copy of the shape in polyt and then rotates and displays it. Repeatedly applying rotation or scaling transformations to an object will usually cause distortion because arithmetic errors will accumulate. By accumulating the transform parameters instead and applying them to an untransformed model, your programs can minimize such distortion.

You can add scaling about a point by making these changes to your program:

```

a=a1: s=1
'-----
FOR r=1 TO 5: FOR c=1 TO 2
  READ path(r,c)
  DATA 4,-4,8,-2,8,2,4,7,2,11
NEXT c,r
'-----
  translate poly(i+1),poly(i),path(m,1),path(m,2)
  polyt(i)=poly(i): polyt(i+1)=poly(i+1)
  IF i>4 THEN Scale polyt(i+1),polyt(i),x0,y0,s,s
  IF i>4 THEN Rotate polyt(i+1),polyt(i),x0,y0,ca,sa
NEXT i
PENPAT 380
FRAMEPOLY VARPTR(pgone(0))
PENPAT 492
FRAMEPOLY VARPTR(polyt(0))
a=a+a1
s=s*.95
'-----
SUB Scale(x,y,x0,y0,s,t) STATIC
  x=x0+s*(x-x0)
  y=y0+t*(y-y0)
END SUB

```

You can use a similar technique to rotate MacPaint images. Keep in mind that rotating every point in a large image requires a tremendous amount of calculation. Don't expect to see fast rotation animation unless you store different views as frames in memory.

The next program loads a MacPaint image in a file created with the module transferral program from Chapter 6. If you wish, you can modify it to read an image from the Clipboard instead. Either way, the program displays the image on the screen, translates it h pixels

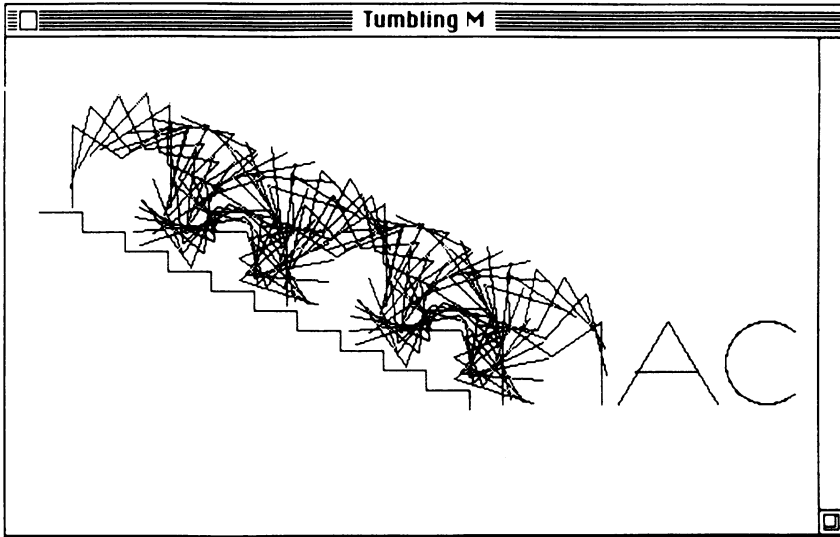


Figure 8-7.
Tumbling letter M

to the right, and rotates it about the midpoint of the GET/PUT rectangle:

```

Animateframes:
f$=FILES$(1)
IF f$="" THEN STOP
OPEN f$ FOR INPUT AS 1
INPUT #1,n,nframe
IF flag=1 THEN ERASE array%
DIM array%(n): flag=1
msg$="loading figure": GOSUB Message
FOR i=0 TO n
  INPUT #1,array%(i)
NEXT i
BUTTON CLOSE 1
CLOSE 1

Rotation:
CLS
INPUT "Enter angle of rotation (-6.28 to 6.28)";a
ca=COS(a): sa=SIN(a)
CLS
msg$="press 's' to stop, 'a' for another angle"

```

```

60SUB message
  ulx=20: uly=50: h=200: k=0
  midx=ulx+h+array%(0)/2: midy=uly+k+array%(1)/2
  PUT (ulx,uly),array%,PSET
  MOVETO ulx+h,uly+k-10
  PRINT "Rotating ";a;"radians";
  FOR row=1 TO array%(1)
    FOR col=1 TO array%(0)
      x=ulx+row: y=uly+col
      IF POINT(x,y)<>33 THEN skip
      Translate x,y,h,k
      Rotate x,y,midx,midy,ca,sa
      PSET(x,y)
    Skip:
    NEXT col
    i$=INKEY$: IF i$="s" OR i$="a" THEN DoKey
  NEXT row
  Stay: i$=INKEY$: IF i$="" THEN Stay
  BUTTON CLOSE 1
  IF i$="s" THEN END
  GOTO Rotation

DoKey:
  BUTTON CLOSE 1
  IF i$="s" THEN END
  IF i$="a" THEN Rotation
END

Message:
  BUTTON 1,1,msg$(20,0)-(470,20)
RETURN

REM ** Subprograms **
SUB Translate(x,y,h,k) STATIC
  x=x+h: y=y+k
END SUB
SUB Rotate(x,y,x0,y0,c,s) STATIC
  xt=x0+(x-x0)*c+(y-y0)*s
  yt=y0+(y-y0)*c-(x-x0)*s
  x=xt: y=yt
END SUB
SUB Scale(x,y,x0,y0,s,t) STATIC
  x=x0+s*(x-x0)
  y=y0+t*(y-y0)
END SUB

```

Note that by using this program, the sample image shown in Figure 8-8 took three minutes to rotate.

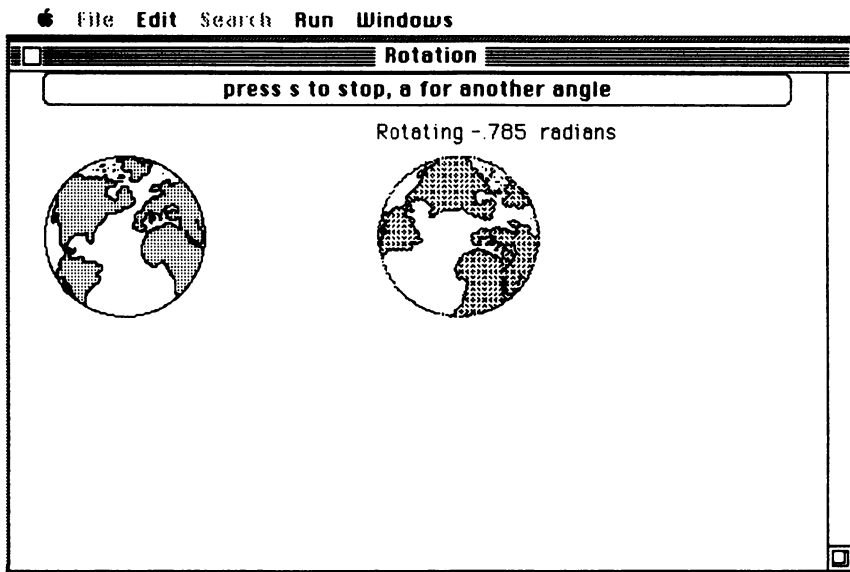


Figure 8-8.
MacPaint image rotated about its midpoint

Three-Dimensional Objects

Representing three-dimensional objects on a two-dimensional computer screen can be a challenge. Until someone develops an inexpensive screen that will display in three dimensions, graphics programmers will have to work around that inherent restriction.

There are several different approaches you can use to project three-dimensional objects onto the screen. In an *isometric* representation, both the x and y axes are offset from the horizontal, and lines that are parallel in three-dimensional objects are drawn parallel, as shown in Figure 8-9.

An *orthogonal* representation shows three different views of the object: top, front, and side. The views may be arranged on the screen as shown in Figure 8-10. The figure also includes an isometric view for comparison.

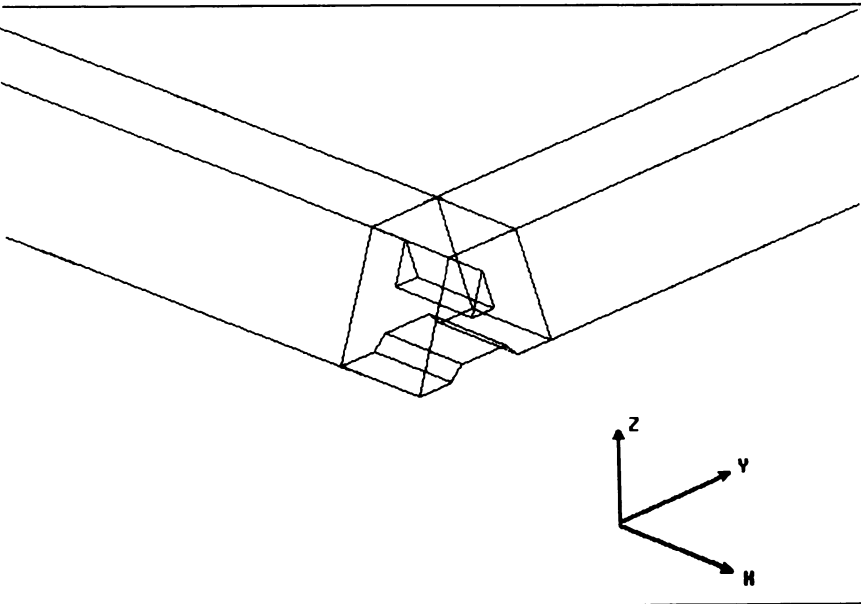


Figure 8-9.
Isometric view of the letter A

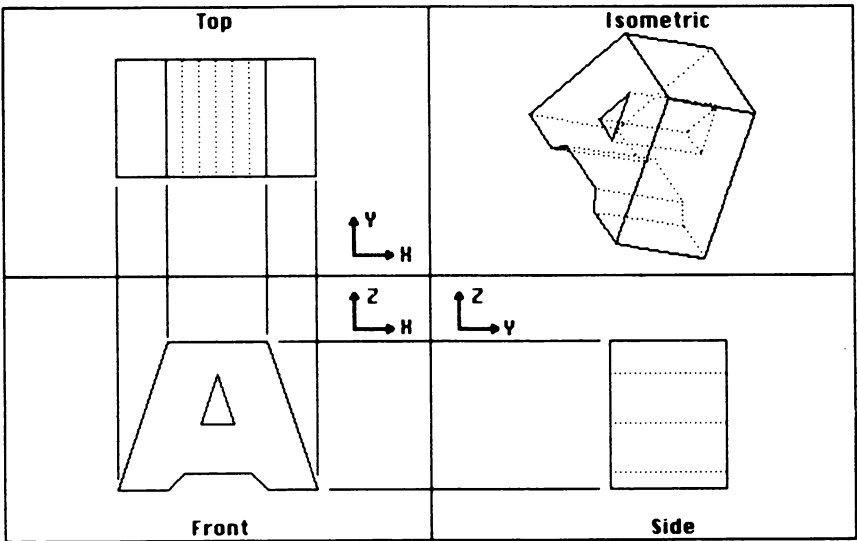


Figure 8-10.
Orthogonal view of the letter A

Draftsmen and artists often use isometric and orthogonal views because they are quick to prepare. But neither of these representations gives a totally accurate depiction of an object. For example, objects that are distant should appear smaller than those that are closer. *Perspective* drawings achieve this effect by causing parallel lines to converge at a distant point, called a *vanishing point*. Figure 8-11 demonstrates a perspective representation.

Perspective representations can be difficult to draw by hand, but they are easy with your Macintosh. The key is to reduce an object's size proportionally to its distance from the viewer. Think of the z axis as extending into the screen. Your program can achieve perspective by dividing each point's x and y coordinate by its z coordinate. Thus, the farther along the z axis a point is (the farther away from the screen), the greater the divisor and therefore the smaller the resulting x and y coordinates.

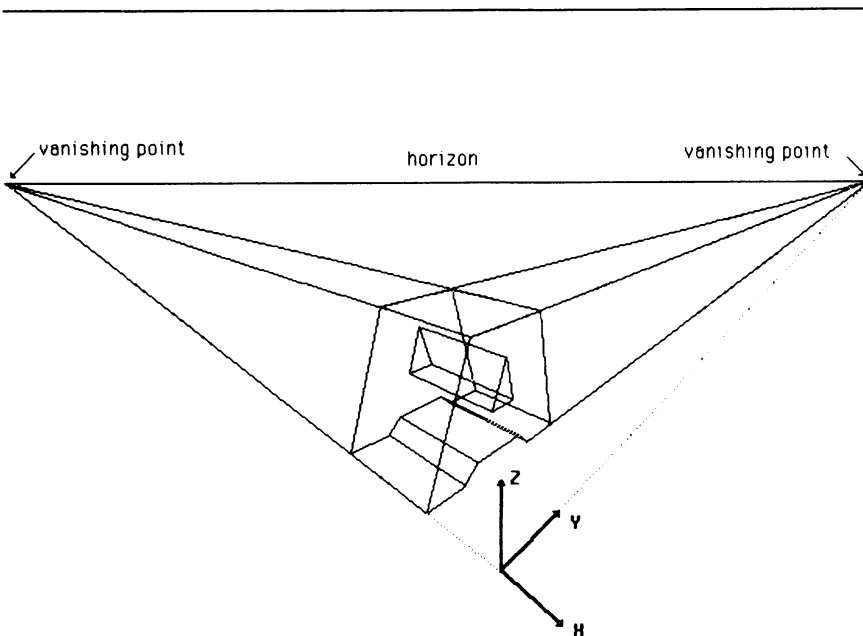


Figure 8-11.
Perspective view of the letter A

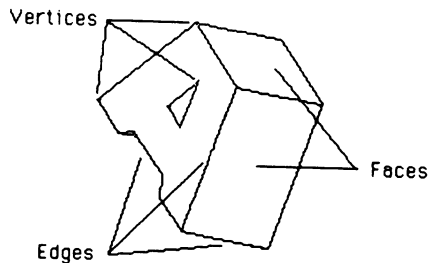
The computer-aided design (CAD) applications program at the end of this chapter uses all three methods to represent three-dimensional objects.

STORING THREE-DIMENSIONAL OBJECTS IN MEMORY

Let's consider how to store a three-dimensional object in memory. Like two-dimensional objects, three-dimensional objects consist of a list of points (vertices). Of course, each such point has one additional coordinate, the distance along the z axis. The notation (x,y,z) is used to represent the three distances along the x, y, and z axes.

To draw the object, you must draw lines connecting the points in the proper order. This kind of display is the simplest to provide. Once drawn, it looks as if the object consists of a web of wires that connect at its corners. This is called a wire-frame representation, sometimes referred to as a schematic drawing.

You must also decide what information to store about each object. As you might suspect, there are many schemes you can use. For some applications, you may wish to store information about the vertices that group to form faces, those that control the color and texture of the faces, and more.



A very simple approach is to store the coordinates for both end-points of each line segment in the object. This is slightly redundant, but it simplifies the program examples. Once you are familiar with the programs in this chapter, you may wish to modify them to eliminate the duplication of vertices.

THREE-DIMENSIONAL TRANSFORMATIONS

The three-dimensional transformations are relatively straightforward extensions of their two-dimensional cousins. Consider rotation, for example. In two dimensions, you only need to consider rotation about a point. To rotate an object in three dimensions, you must specify angles of rotation about all three axes. Our programs could then rotate a point about each axis by using the two-dimensional technique three times. There is, however, a more efficient method. The programs in this chapter combine all three rotations into a single matrix multiplication operation.

Three-dimensional scaling and translation are nearly identical to their two-dimensional counterparts. In translation, you merely add x, y, and z distances to the corresponding coordinates of each point. For scaling, multiply each coordinate by the corresponding scaling factor.

STORING AND MANIPULATING THREE-DIMENSIONAL OBJECTS

The next program stores and manipulates a three-dimensional object. The program reads the endpoint coordinate values into six arrays (three arrays for each of the endpoints) and then asks you to select a viewing mode (perspective or isometric), a scaling factor (1 = no change, less than 1 = reduction, greater than 1 = expansion), and rotation angles about each of the three coordinate axes (in degrees):

```

GOSUB InitSys
GOSUB LoadData
Main0:
GOSUB Viewm
WHILE MOUSE(0)>=0: WEND
GOTO main0
.
.   draw a 3D-line to one quadrant
.
DrawLine:
  x1=xw1(lptr)*rotmat(1,1)+yw1(lptr)*rotmat(1,2)+zw1(lptr)*rotmat(1,3)
  y1=xw1(lptr)*rotmat(2,1)+yw1(lptr)*rotmat(2,2)+zw1(lptr)*rotmat(2,3)
  x2=xw2(lptr)*rotmat(1,1)+yw2(lptr)*rotmat(1,2)+zw2(lptr)*rotmat(1,3)
  y2=xw2(lptr)*rotmat(2,1)+yw2(lptr)*rotmat(2,2)+zw2(lptr)*rotmat(2,3)
  x1=x1*scale:x2=x2*scale
  y1=y1*scale:y2=y2*scale

```

```

IF NOT persp THEN DrawLine0
z1=xw1(lptr)*rotmat(3,1)+yw1(lptr)*rotmat(3,2)+zw1(lptr)*rotmat(3,3)
z2=xw2(lptr)*rotmat(3,1)+yw2(lptr)*rotmat(3,2)+zw2(lptr)*rotmat(3,3)
z1=z1*scale;z2=z2*scale
t1=250/(250+z1);t2=250/(250+z2)
y1=y1*t1;y2=y2*t2
x1=x1*t1;x2=x2*t2
DrawLine0:
LINE(xorg+x1,yorg-y1)-(xorg+x2,yorg-y2),color
RETURN

```

```

' build the rotation matrix
'

```

```

BuildMat:
rotmat(1,1)=COS(zrot)*COS(yrot)
rotmat(1,2)=SIN(zrot)*COS(yrot)
rotmat(1,3)=-SIN(yrot)
rotmat(2,1)=-SIN(zrot)*COS(xrot)+
COS(zrot)*SIN(yrot)*SIN(xrot)
rotmat(2,2)=COS(zrot)*COS(xrot)+
SIN(zrot)*SIN(yrot)*SIN(xrot)
rotmat(2,3)=COS(yrot)*SIN(xrot)
rotmat(3,1)=SIN(zrot)*SIN(xrot)+
COS(zrot)*SIN(yrot)*COS(xrot)
rotmat(3,2)=-COS(zrot)*SIN(xrot)+
SIN(zrot)*SIN(yrot)*COS(xrot)
rotmat(3,3)=COS(yrot)*COS(xrot)
RETURN

```

```

LoadData:
numlin=33
FOR i=1 TO numlin
  READ xw1(i),yw1(i),zw1(i)
  READ xw2(i),yw2(i),zw2(i)
NEXT i
DATA -60,-40,-50,-30,-40,-50
DATA -30,-40,-50,-20,-40,-40
DATA -20,-40,-40,20,-40,-40
DATA 20,-40,-40,30,-40,-50
DATA 30,-40,-50,60,-40,-50
DATA 60,-40,-50,30,-40,40
DATA 30,-40,40,-30,-40,40
DATA -30,-40,40,-60,-40,-50
DATA -10,-40,-10,10,-40,-10
DATA 10,-40,-10,0,-40,20
DATA 0,-40,20,-10,-40,-10
DATA -60,30,-50,-30,30,-50
DATA -30,30,-50,-20,30,-40
DATA -20,30,-40,20,30,-40
DATA 20,30,-40,30,30,-50

```



```

DATA 30,30,-50,60,30,-50
DATA 60,30,-50,30,30,40
DATA 30,30,40,-30,30,40
DATA -30,30,40,-60,30,-50
DATA -10,30,-10,10,30,-10
DATA 10,30,-10,0,30,20
DATA 0,30,20,-10,30,-10
DATA -60,-40,-50,-60,30,-50
DATA -30,-40,-50,-30,30,-50
DATA -20,-40,-40,-20,30,-40
DATA 20,-40,-40,20,30,-40
DATA 30,-40,-50,30,30,-50
DATA 60,-40,-50,60,30,-50
DATA 30,-40,40,30,30,40
DATA -30,-40,40,-30,30,40
DATA -10,-40,-10,-10,30,-10
DATA 10,-40,-10,10,30,-10
DATA 0,-40,20,0,30,20
RETURN

```

Viewm:

```

GOSUB OpWin
MOVETO 10,15:PRINT"Scale:"
MOVETO 130,15:PRINT"X rot:"
MOVETO 10,35:PRINT"Y rot:"
MOVETO 130,35:PRINT"Z rot:"
EDIT FIELD 4,STR$(zr),(180,25)-(235,40),1,3
EDIT FIELD 3,STR$(yr),(60,25)-(115,40),1,3
EDIT FIELD 2,STR$(xr),(180,5)-(235,20),1,3
EDIT FIELD 1,STR$(scale),(60,5)-(115,20),1,3
BUTTON 2,2,"Perspective",(1,45)-(119,59)
BUTTON 1,2,"Isometric",(120,45)-(239,59)
dial0=DIALOG(0):dial0=DIALOG(0)
i=1
Expan0:
dial0=DIALOG(0)
IF (dial0=0) OR (dial0=2) THEN Expan0
IF dial0<5 THEN expan1
i=i+1+(i>3)*4
EDIT FIELD i
GOTO Expan0
Expan1:
IF (dial0=1) AND (DIALOG(1)=2) THEN persp=true ELSE persp=false
scale=VAL(EDIT$(1))
xr=VAL(EDIT$(2)):xrot=xr/57.3
yr=VAL(EDIT$(3)):yrot=yr/57.3
zr=VAL(EDIT$(4)):zrot=zr/57.3
GOSUB buildmat
xorg=256:yorg=160
WINDOW CLOSE 2
color=33

```

```

CLS
IF numlin=0 THEN RETURN
FOR lptr=1 TO numlin
  GOSUB DrawLine
NEXT lptr
RETURN
'
' open up the message window
'
OpWin:
GET (135,128)-(375,190),bitsev
WINDOW 2,,(135,150)-(375,210),3
RETURN
'
' close message window
'
ClosWin:
WINDOW CLOSE 2
PUT(135,128),bitsev,PSET
RETURN
'
' initialize the system at startup
'
InitSys:
DEFINT a-n
DEFSNG o-z
DIM xw1(200),xw2(200),yw1(200),yw2(200),zw1(200)
DIM zw2(200),rotmat(3,3),bitsev(1010)
true=-1:false=0
numlin=0:scale=1
WINDOW 1,,(0,20)-(511,341),3
RETURN

```

The Dralin routine draws a line. It begins by rotating and scaling the x and y coordinates. If the user requests a perspective drawing, it rotates and scales the z coordinates and uses them to adjust x and y, creating the perspective effect. After all the calculations are complete, it draws the line between the resulting x and y values.

The BuildMat routine computes the rotation matrix. The user supplies the three rotation angles xrot, yrot, and zrot. BuildMat uses these numbers to calculate nine values stored in the RotMat array. The Dralin routine uses these values to rotate the x and y coordinates.

The Viewm routine builds an edit dialog box. It gets values from the user and then calls the BuildMat and Dralin routines to rotate, scale, and draw the figure.

Computer-Aided Design Program

Computer-aided design (CAD) is the process of using computers to assist in product design. The need for efficient ways to design everything from nuts and bolts to automobiles and spacecraft has been one of the primary driving forces in the development of computer graphics. Traditionally, CAD programs reside on mainframes and minicomputers because of memory and processing-speed requirements. Nevertheless, you can implement a simple CAD system on your Macintosh by using—of all languages—BASIC.

Admittedly, our CAD program is not going to put any software developers out of business. It does demonstrate, however, a vital application of the routines and concepts developed in this chapter. Typing a program of this length is not for the faint-hearted. If you wish to save some typing time, you can send away for the program disk described on page ix.

· this program creates and displays 3D graphic images

```

GOSUB InitSys
GOSUB UpDate
WHILE true
  WHILE MOUSE(0)<1:MENU ON
    IF (flagv=1 OR MOUSE(1)>256 AND MOUSE(2)<140 ) AND (numlin <> 0 ) AND
      (updateview)THEN GO SUB Scan
  WEND
  MENU STOP
  GOSUB AddAPoint
  MENU ON
WEND
END

```

· add another point

```

AddAPoint:
  numlin=numlin+1
  IF numlin=1 THEN MENU 5,2,1: MENU 5,3,1
  xw1(numlin)=xw1:yw1(numlin)=yw1:zw1(numlin)=zw1
  GOSUB Get3d
  IF cancel THEN numlin=numlin-1:RETURN
  xw2(numlin)=xw1:yw2(numlin)=yw1:zw2(numlin)=zw1
  color=33:1stxt=numlin:1stp=numlin
  GOSUB UDetRange
RETURN

```

· update the screen

UpDate:

```

MENU 4,1,1
selected =false
updateview=true
CLS
LINE(0,160)-(511,160)
LINE(256,0)-(256,321)
IF !okflg THEN UpDate0
FOR dx=-120 TO 120 STEP 10
  FOR dy=-80 TO 80 STEP 10
    PSET(128+dx,80+dy)
    PSET(128+dx,240+dy)
    PSET(384+dx,240+dy)
  NEXT dy
NEXT dx
UpDate0:
IF numlin=0 THEN RETURN
color=33
lstrt=1
lstp=numlin
GOSUB UDatRange
RETURN

```

· update a range of line records

UDatRange:

```

persp=false
scale=1
xrot=0
yrot=0
zrot=0
xorg=128:yorg=80
GOSUB BuildMat
FOR lptr=lstrt TO lstp
  GOSUB DraLin
NEXT lptr
xrot=1.571
yrot=0
zrot=0
xorg=128:yorg=240
GOSUB BuildMat
FOR lptr=lstrt TO lstp
  GOSUB DraLin
NEXT lptr
xrot=1.571
yrot=0
zrot=1.571
xorg=384:yorg=240
GOSUB BuildMat

```

```

FOR lptr=lstl TO lstp
  GOSUB DraLin
NEXT lptr
xrot=1.047
yrot=5.76
zrot=1.047
xorg=384:yorg=80
GOSUB BuildMat
FOR lptr=lstl TO lstp
  GOSUB DraLin
NEXT lptr
RETURN
'
' draw a 3D-line to one quadrant
'
DraLin:
x1=xw1(lptr)*rotmat(1,1)+yw1(lptr)*rotmat(1,2)+zw1(lptr)*rotmat(1,3)
y1=xw1(lptr)*rotmat(2,1)+yw1(lptr)*rotmat(2,2)+zw1(lptr)*rotmat(2,3)
x2=xw2(lptr)*rotmat(1,1)+yw2(lptr)*rotmat(1,2)+zw2(lptr)*rotmat(1,3)
y2=xw2(lptr)*rotmat(2,1)+yw2(lptr)*rotmat(2,2)+zw2(lptr)*rotmat(2,3)
x1=x1*scale:x2=x2*scale
y1=y1*scale:y2=y2*scale
IF NOT persp THEN DraLin0
z1=xw1(lptr)*rotmat(3,1)+yw1(lptr)*rotmat(3,2)+zw1(lptr)*rotmat(3,3)
z2=xw2(lptr)*rotmat(3,1)+yw2(lptr)*rotmat(3,2)+zw2(lptr)*rotmat(3,3)
z1=z1*scale:z2=z2*scale
t1=250/(250+z1):t2=250/(250+z2)
y1=y1*t1:y2=y2*t2
x1=x1*t1:x2=x2*t2
DraLin0:
LINE(xorg+x1,yorg-y1)-(xorg+x2,yorg-y2),color
RETURN
'
' build the rotation matrix
'
BuildMat:
rotmat(1,1)=COS(zrot)*COS(yrot)
rotmat(1,2)=SIN(zrot)*COS(yrot)
rotmat(1,3)=-SIN(yrot)
rotmat(2,1)=-SIN(zrot)*COS(xrot)+COS(zrot)*SIN(yrot)*SIN(xrot)
rotmat(2,2)=COS(zrot)*COS(xrot)+SIN(zrot)*SIN(yrot)*SIN(xrot)
rotmat(2,3)=COS(yrot)*SIN(xrot)
rotmat(3,1)=SIN(zrot)*SIN(xrot)+COS(zrot)*SIN(yrot)*COS(xrot)
rotmat(3,2)=-COS(zrot)*SIN(xrot)+SIN(zrot)*SIN(yrot)*COS(xrot)
rotmat(3,3)=COS(yrot)*COS(xrot)
RETURN
'
' get a 3D cursor point
'
Get3d:
IF updateview THEN Get3dA
GOSUB OpWin

```

```

PRINT
PRINT "You must return to the update view"
PRINT " before you can select a point"
cancel = true
FOR j=1 TO 20000 : NEXT j
GOSUB ClosWin
RETURN
Get3dA:
GOSUB Get2d
i=true
FOR j=1 TO 3
  IF NOT(setdpth(j) OR setcoor(j)) THEN i=false
NEXT j
IF i THEN Get3dB
GOSUB OpWin
PRINT" Point is ambiguous. Select depth"
PRINT " in another quadrant and try again"
GOSUB ChkUsr
IF cancel THEN RETURN
WHILE MOUSE(0)<1:WEND
GOTO Get3d
Get3dB:
IF setcoor(1) THEN xw1=xw ELSE xw1=xdpth
IF setcoor(2) THEN yw1=yw ELSE yw1=ydpth
IF setcoor(3) THEN zw1=zw ELSE zw1=zdpth
RETURN

'
'MENU command
'

ProcessM:
menuid=MENU(0)
itemid=MENU(1)
FOR pmi=1 TO 5
  MENU pmi,0,0
NEXT
MENU OFF

ON menuid GOSUB FileM,ViewM,LockM,DepthM,Prim
FOR pmi=1 TO 5
  MENU pmi,0,1
NEXT
IF NOT updateview THEN MENU 4,0,0 : MENU 5,0,0
MENU ON
ON MENU GOSUB ProcessM
RETURN

' file command

FileM:
ON itemid GOTO FilSav,FilLoad,FilQuit

```

```

FilSav:
n$=FILES$(0,filnam$)
IF (numlin=0) OR (LEN(n$)=0) THEN FilSav0
filnam$=n$
OPEN filnam$ FOR OUTPUT AS 1
FOR i=1 TO numlin
PRINT# 1,xw1(i)
PRINT# 1,xw2(i)
PRINT# 1,yw1(i)
PRINT# 1,yw2(i)
PRINT# 1,zw1(i)
PRINT# 1,zw2(i)
NEXT i
CLOSE 1
FilSav0:
GOSUB UpDate
RETURN
    
```

```

FilLoad:
n$=FILES$(1)
IF LEN(n$)=0 THEN GOSUB UpDate:RETURN
filnam$=n$
OPEN filnam$ FOR INPUT AS 1
numlin=0
FilLoad0:
IF EOF(1) THEN CLOSE 1:GOSUB UpDate: RETURN
numlin=numlin+1
INPUT# 1,xw1(numlin)
INPUT# 1,xw2(numlin)
INPUT# 1,yw1(numlin)
INPUT# 1,yw2(numlin)
INPUT# 1,zw1(numlin)
INPUT# 1,zw2(numlin)
GOTO FilLoad0
    
```

```

FilQuit:
GOSUB OpWin
PRINT " Are sure that you "
PRINT " want to stop?"
GOSUB ChkUsr
IF cancel THEN RETURN
MENU RESET
END
    
```

```

' draw a view
    
```

```

ViewM:
ON itemid GOTO UpDate,Expand
    
```

```

Expand:
MENU 4,0,0: MENU 5,0,0
    
```

```

flagv=1
GOSUB OpWin
MOVETO 10,15:PRINT"Scale:"
MOVETO 130,15:PRINT"X rot:"
MOVETO 10,35:PRINT"Y rot:"
MOVETO 130,35:PRINT"Z rot:"
EDIT FIELD 4,STR$(zr),(180,25)-(235,40),1,3
EDIT FIELD 3,STR$(yr),(60,25)-(115,40),1,3
EDIT FIELD 2,STR$(xr),(180,5)-(235,20),1,3
EDIT FIELD 1,STR$(scale),(60,5)-(115,20),1,3
BUTTON 2,2,"Perspective",(1,45)-(119,59)
BUTTON 1,2,"Isometric",(120,45)-(239,59)
dia10=DIALOG(0):dia10=DIALOG(0)
i=1
updateview=false
Expan0:
dia10=DIALOG(0)
IF (dia10=0) OR (dia10=2) THEN Expan0
IF dia10<5 THEN Expan1
i=i+1+(i>3)*4
EDIT FIELD i
GOTO Expan0
Expan1:
IF (dia10=1) AND (DIALOG(1)=2) THEN persp=true ELSE persp=false
scale=VAL(EDIT$(1))
xr=VAL(EDIT$(2)):xrot=xr/57.3
yr=VAL(EDIT$(3)):yrot=yr/57.3
zr=VAL(EDIT$(4)):zrot=zr/57.3
GOSUB BuildMat
xorg=256:yorg=160
WINDOW CLOSE 2
color=33
CLS
IF numlin=0 THEN RETURN
FOR lptr=1 TO numlin
  GOSUB DraLin
NEXT lptr
RETURN

' toggle the grid lock

LockM:
lokflg=1-itemid
MENU 3,itemid,2
MENU 3,2+lokflg,1
RETURN

' set new active depth

DepthM:
i=MOUSE(0)

```



```

WHILE MOUSE(0)<1:WEND
GOSUB Get2d
IF cancel THEN RETURN
FOR i=1 TO 3
    setdpth(i)=setcoor(i)
NEXT i
xdpth=xw:ydpth=yw:zdpth=zw
RETURN
    
```

' get 2D cursor point

```

Get2d:
cancel=false
xs=MOUSE(1):ys=MOUSE(2)
IF expanded = 1 THEN GOSUB NoDraw: RETURN
IF ys>160 THEN get2d1
IF xs<256 THEN get2d0
GOSUB OpWin
PRINT " Sorry, you can't click here."
PRINT "This window is for display only."
GOSUB ChkUsr
IF cancel THEN RETURN
WHILE MOUSE(0)<1:WEND:GOSUB Get2d:RETURN
Get2d0:
setcoor(1)=true:xw=xs-128
setcoor(2)=true:yw=80-ys
setcoor(3)=false
GOTO get2dex
Get2d1:
setcoor(3)=true:zw=240-ys
IF xs<256 THEN get2d2
setcoor(1)=false
setcoor(2)=true:yw=xs-384
GOTO Get2dEx
Get2d2:
setcoor(1)=true:xw=xs-128
setcoor(2)=false
Get2dEx:
IF lokflg THEN RETURN
xw=INT((xw+5)/10)*10
yw=INT((yw+5)/10)*10
zw=INT((zw+5)/10)*10
RETURN
    
```

' perform a graphic primitive function

```

Prim:
ON itemid GOTO Prim0,DelLast,DelAll
    
```

```

Prim0:
WHILE MOUSE(0)<1:WEND
IF expanded =1 THEN GOSUB NoDraw: RETURN
    
```

GOSUB Get3d

RETURN

' delete the last line from the data base

DelLast:

IF numlin=0 THEN RETURN

lstrt=numlin

lstp=numlin

color=30

GOSUB UDatRange

numlin=numlin-1

xw1=xw2(numlin):yw1=yw2(numlin):zw1=zw2(numlin)

RETURN

' delete the entire graphic data base

DelAll:

GOSUB OpWin

PRINT " Do you want to delete "

PRINT " the entire picture?"

GOSUB ChkUsr

IF cancel THEN RETURN

numlin=0

GOSUB UpDate

RETURN

' check with user before execution

ChkUsr:

BUTTON 2,2,"Cancel",(130,40)-(199,59)

BUTTON 1,2,"Ok",(200,40)-(239,59)

dia10=**DIALOG(0):dia10=DIALOG(0)**

ChkUntil:

dia10=**DIALOG(0)**

IF dia10<>1 THEN ChkUntil

IF DIALOG(1)=2 THEN cancel=true ELSE cancel=false

GOSUB ClosWin

RETURN

' open up the message window

OpWin:

GET (135,128)-(375,190),bitsev

WINDOW 2,,(135,150)-(375,210),3

RETURN

' close message window

ClosWin:

WINDOW CLOSE 2

PUT(135,128),bitsev,PSET

RETURN

initialize the system at startup

InitSys:

```

DEFINT a-n
DEFSNG o-z
DIM xw1(200),xw2(200),yw1(200),yw2(200),zw1(200),zw2(200)
DIM rotmat(3,3),bitsav(1010),setcoor(3),setdpth(3),cursor(33),color(200)
true=-1:false=0
filnam$=""
numlin=0:scale=1
selected = false
MENU 1,0,1,"File"
  MENU 1,1,1,"save"
  MENU 1,2,1,"load"
  MENU 1,3,1,"stop"
MENU 2,0,1,"View"
  MENU 2,1,1,"update screen"
  MENU 2,2,1,"expanded view"
MENU 3,0,1,"Lock"
  MENU 3,1,2,"grid on"
  MENU 3,2,1,"grid off"
MENU 4,0,1,"Depth"
  MENU 4,1,1,"active"
MENU 5,0,1,"Primitives"
  MENU 5,1,1,"new stream"
  MENU 5,2,0,"delete last"
  MENU 5,3,0,"delete all"
WINDOW 1,,(0,20)-(511,341),3
ON MENU GOSUB ProcessM
MENU ON
' Input Cursor:
FOR i=0 TO 33: READ cursor(i) : NEXT i
DATA 10752,7168,-128,7168,10752,18688,-30592,0,0,0,0
DATA 0,0,0,0,0,10752,7168,-2176,7168,10752,18688,-30592
DATA 0,0,0,0,0,0,0,0,0,2,4
RETURN

```

NoDepth:

```

GOSUB OpWin
PRINT "Sorry, you have to go to the updated"
PRINT " screen to access this function."
FOR j=1 TO 20000: NEXT j
GOSUB ClosWin
RETURN

```

NoPrim:

```

GOSUB OpWin
PRINT "Sorry, you have to go to the"
PRINT "updated screen to access this function"
FOR j=1 TO 20000: NEXT j
GOSUB ClosWin

```

RETURN

Scan:

MENU ON: MENU STOP**IF MOUSE(1)<257 OR MOUSE(2)>159 THEN INITCURSOR : RETURN****IF POINT(MOUSE(1),MOUSE(2))=33 THEN SETCURSOR VARPTR(cursor(0)) ELSE INITCURSOR****IF MOUSE(0)>=0 THEN Scan****mx = MOUSE(1) : my = MOUSE(2)****IF POINT(mx,my)=33 THEN INITCURSOR: GOSUB UnWantedLine: GOTO Scan****GET(240,1)-(420,30),bitsev****LOCATE 1,40****PRINT "not on line"****FOR z=1 TO 500: NEXT z****PUT(240,1)-(420,30),bitsev,PSET****GOTO Scan****RETURN**

* Find coordinates of unwanted line

UnWantedLine:

cleardialog=DIALOG(0)**FOR lptr=1 TO numlin****GOSUB DraLin****x1=xorg+x1: y1=yorg-y1****x2=xorg+x2: y2=yorg-y2****IF x2 <> x1 THEN notvertical****IF mx = x1 AND ABS(y1-my) < ABS(y1-y2) THEN GOTO Remove ELSE CheckNext****NotVertical:****vabs=ABS(my-y1-(y2-y1)/(x2-x1)*(mx-x1))****IF vabs<1 THEN remove****CheckNext:****NEXT lptr****GET(240,1)-(420,30),bitsev****LOCATE 1,40****PRINT "not on line"****FOR z=1 TO 500: NEXT z****PUT(240,1)-(420,30),bitsev,PSET****RETURN**

* Erase hidden line)

Remove:

GET(115,290)-(315,320),bitsev**BUTTON 1,2,"Delete?",(120,295)-(210,314)****BUTTON 2,2,"Leave alone?",(220,295)-(310,314)****Query:****FOR i=1 TO 500: NEXT i****LOCATE 1,1:****LINE(x1,y1)-(x2,y2),30****FOR i=1 TO 500: NEXT i****LINE(x1,y1)-(x2,y2),33****IF DIALOG(0)=0 THEN Query**

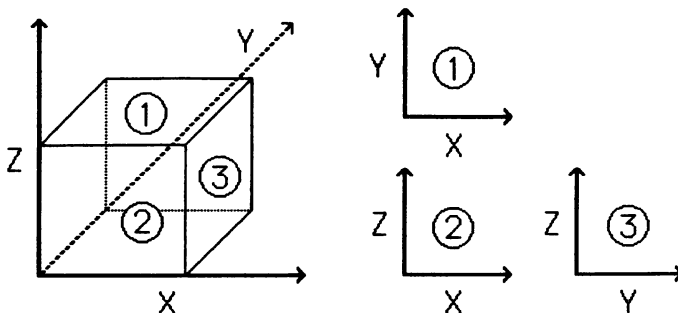
```

BUTTON CLOSE 1
BUTTON CLOSE 2
PUT (115,290)-(315,320),bitsev,PSET
IF DIALOG(1)=2 THEN RETURN
LINE(x1,y1)-(x2,y2),30
FOR i=1ptr TO numlin-1
  xw1(i)=xw1(i+1): xw2(i)=xw2(i+1)
  yw1(i)=yw1(i+1): yw2(i)=yw2(i+1)
  zw1(i)=zw1(i+1): zw2(i)=zw2(i+1)
NEXT i
  numlin=numlin-1
RETURN

```

The CAD program displays an orthogonal representation of a wire-frame object of your design, as well as an isometric representation of the object in the upper-right quadrant. To start the program, select Active from the Depth menu. The menu bar dims to let you know that you should click a point in one of the three quadrants displaying an orthogonal view—upper left (top view), lower left (front view), or lower right (side view). This establishes two coordinate values (for example, x and y) for a point in the upper-left quadrant. Next, select New Stream from the Primitives menu. Now you are ready to select points.

Choose a point in one of the other two orthogonal-view quadrants (not the one you clicked after you activated the Depth menu). Let's say you click a point in the xz quadrant (lower left). That sets the x and z values for a three-dimensional point (x,y,z) . The y value is supplied by the y coordinate of the point that you clicked for the active depth. Continue clicking more points in that quadrant. The program draws a line for each new point. It connects all the lines until you select New Stream from the Primitives menu. New Stream starts a new stream of connected points:



When you are ready to work in a different plane, simply select a new Depth plane and a New Stream. If you don't like the line you have just drawn, select Delete Last from the Primitives menu. If you decide to scrap it all, select Delete All from the Primitives menu.

You can also delete a selected line by using the mouse. Move the cursor into the upper-right quadrant and position it over the line you wish to delete. Click the button when the cursor changes shape. The program tests each pair of points in your data base to see if the point you clicked is approximately on one of the lines. If a match is made, the line flashes. You can then click Delete to delete it or Leave to leave it in the data base.

The Lock menu lets you disable the grid so you can position the points more precisely. If the screen gets messed up in any way, selecting Update Screen from the View menu redraws the object.

If you wish to view the object on the whole screen, select Expanded View from the View menu. Expanded View displays a dialog box that you may recognize from the previous program. You can scale the figure, set rotation values for each of the three axes, and choose either an isometric or perspective view. You can select Expanded View several times to try different view angles and scale factors.

When you are ready to continue editing the figure, select Update Screen again from the View menu. This will return you to the previously displayed orthogonal representation.

When your creation is ready to be immortalized, select Save from the File menu. The Load option of the File menu lets you load your creations from disk.

Summary

The programs developed in this chapter demonstrate transformations of objects on the computer screen. The transformations include translations, rotations, and scaling. Transformation of three-dimensional objects requires the additional consideration of how to project the objects onto a two-dimensional screen. Orthogonal, isometric, and perspective representations of three-dimensional objects have been discussed in this chapter.

In the next chapter, we will explore some techniques for writing efficient programs. You will learn how to 'shoehorn' an overgrown

BASIC program into a 128K Mac and how to speed up those sluggish bits of code.

BASIC Statements

SUB...STATIC

END SUB

9

Designing Efficient Programs

Coding a computer program is not an exact science. Although a programmer must follow the basic rules of the language, program design is largely a matter of personal style. The way you write a program also depends on the results you want: you may want one program to be a model of structured design and well-documented code, another to use the least possible memory, and another to be optimized for fast execution. It is not always possible for a program to do all three equally well, so you must analyze your needs and establish priorities. In this chapter we discuss ways to design programs with efficient use of memory and increased program speed.

Working with Limited Memory

Manipulating complex graphics images can require large amounts of memory. If you use a 128K system, you may find your programs severely restricted unless you pay close attention to memory usage. Fortunately, there are several tools and techniques you can use to work within the memory limitations of your system.

MEMORY PARTITIONS

We'll start with a brief look at how the Mac uses memory. Microsoft BASIC on the Macintosh separates available memory into three distinct areas, called the stack, the data segment, and the heap.

The *stack* keeps track of program flow when you use nested loops and calls to subprograms and subroutines. The deeper the level of nesting in your programs, the more stack space they will require.

The *data segment* contains the program text, along with string variables, numeric variables, and file buffers for opened files.

The *heap*, as its name implies, contains a heap of goodies. First and foremost, it contains the active segments of the BASIC program itself. Instead of all of the more than 80K of BASIC code being loaded into memory at once, sections of the language are loaded into memory as they are needed. This can slow down program execution when new segments are loaded into memory, but it keeps the net amount of memory required by BASIC to a tolerable level. The heap also contains a 1024 byte SOUND buffer, image data recorded by the PICTURE statement, and storage for such user interface structures as buttons, edit fields, and active desktop objects.

CONTROLLING MEMORY ALLOCATION

Each of these sections is allocated a predetermined amount of memory, depending on how much memory your Macintosh has. On a 512K system, Binary BASIC 2.0 allocates around 52K bytes for the heap, 14K for the stack, and 334K for the data segment. On a 128K system, Binary BASIC allocates 13K for the heap, 6K for the stack, and 20K for the data segment.

If you want to change these default values, BASIC provides a means to control memory allocation via the CLEAR statement. This statement completely resets the stack and all variables, functions, and string space to null or zero status. The format is

```
CLEAR,data,stack
```

where data is the number of bytes reserved for BASIC's data segment and stack is the number of bytes reserved for the stack. Both the data and stack parameters are optional; if not supplied, a parameter defaults to its current value. The heap space is whatever is left over after the data and stack areas are set. If you are doing a lot of bit image array manipulation in a program, you may need to increase the size of the data segment and decrease the size of the stack.

MONITORING MEMORY ALLOCATION

To help you monitor the amount of available memory, BASIC provides the function FRE(x). This function returns the amount of unused heap space if x is -1, the amount of unused stack space if x is -2, and the amount of unused memory in the data segment if x is any other number. If you have large amounts of data and you are pushing the limits of available memory, use the FRE function to monitor the memory demands made by your program.

Keep in mind that the values returned by FRE give an instantaneous assessment of free memory; these values change continuously while your programs run, as demonstrated by this listing:

```
h=FRE(-1): PRINT "heap:";h
PRINT FRE (0)
s=FRE(-2): PRINT "stack:";s
PRINT FRE (0)
d=FRE(0): PRINT "data:";d
PRINT FRE (0)
t=h+s+d: PRINT "total:";t
PRINT FRE (0)
z: IF INKEY$="" THEN z
```

Each time your program introduces a new variable, it reduces the data segment memory by a minimum of eight bytes:

```
heap: 13490
20820
stack: 6323
20812
data: 20804
20804
total: 40617
20796
```

Remember, the default data type for Binary BASIC is single-precision, which requires four data bytes per variable plus extra memory for the variable's name and other overhead. (Integers consume two data bytes, and double-precision variables take eight.)

Designing for Limited Memory

There are several ways to condense a program so that it takes up less room. Users with 128K Macs may be forced to sacrifice some program readability to make the most of the memory available.

Keep the use of line numbers, labels, and remarks to an absolute minimum (see Figure 9-1). Use short variable names. You can also place multiple statements on a single line and remove indentations (as in nested loops).

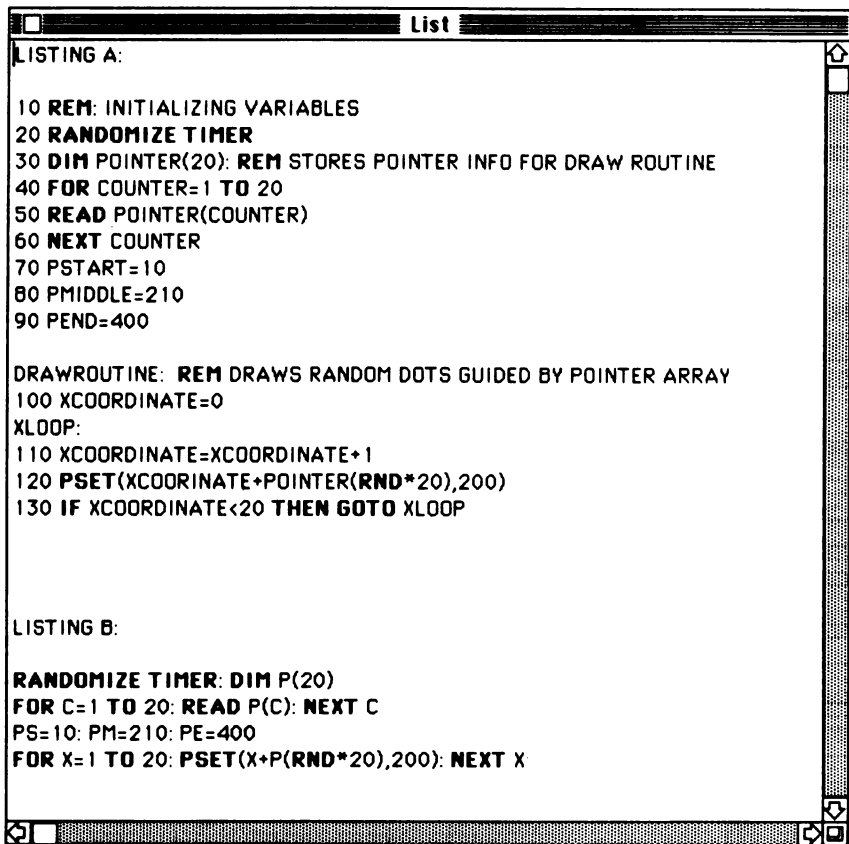


Figure 9-1.
Examples of ways to condense programs

Sometimes you can reuse the same variable for several purposes, rather than introducing a flock of new variables (see Figure 9-2). This cuts down on variable storage.

Another trick is not to use the optional verb CALL when using ROM routines (see Figure 9-3).

Remember, use these techniques only as a last resort. Eliminating comments and reusing variables can make a program exceedingly difficult to understand, especially if the program sits idle for a month or two.

CHAINING AND OVERLAYING PROGRAMS

Another way you can make room for variable storage is to decrease the instantaneous size of your program. For example, assume you have written a magnificent music composition tool. Suppose it consists of two main sections: one helps the user compose, and the other plays the resulting music. You can take advantage of the fact that these two program sections need not be in memory at the same time. The composition section merely passes control to the playing section when necessary. When the playing section is finished, it passes control back to the composition section. In BASIC, program control is passed from one segment to the next with the CHAIN statement.

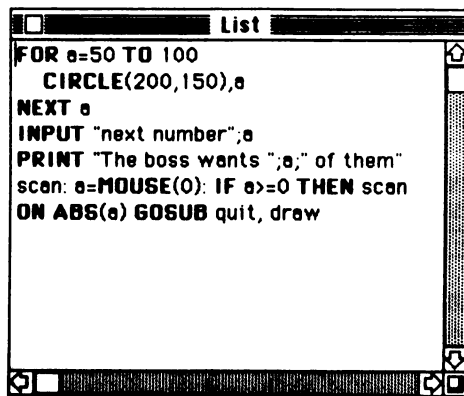


Figure 9-2.
Reusing variables

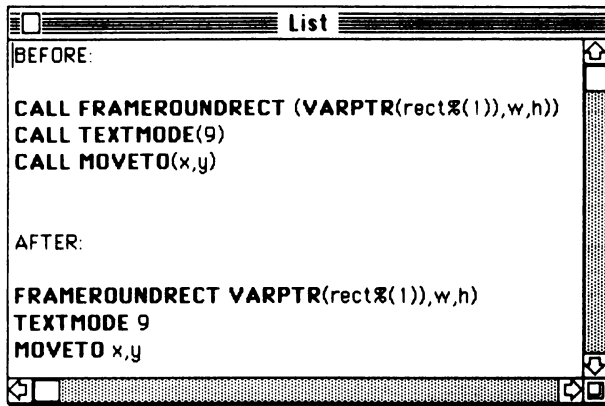


Figure 9-3.
Do not use CALL

When BASIC executes a CHAIN statement, it merely replaces the current program with a new one and begins executing it. If the program segments need to share certain variables, include the variable names in a COMMON statement at the beginning of both segments. You can also tell BASIC to share all variables with CHAIN's All option.

A related technique is called *overlaying*. Overlays are used when several sections of your program share some code. When you use overlays, the same program continues to run, but you replace parts of it as necessary. The CHAIN MERGE statement helps you do this. You specify the range of statements to replace, and BASIC does the rest. Figure 9-4 contains a skeletal version of an overlaid program. The main program replaces labels s through e with overlay "inside" or "outside" as necessary.

RECLAIMING ARRAY MEMORY

Array variables allow programmers to reference a group of variables of the same type with a single variable name. Before using an array, you must reserve memory by using the DIM statement. If your program uses an array for one portion of the program run and never uses it again, you can reclaim the space it occupied with the ERASE

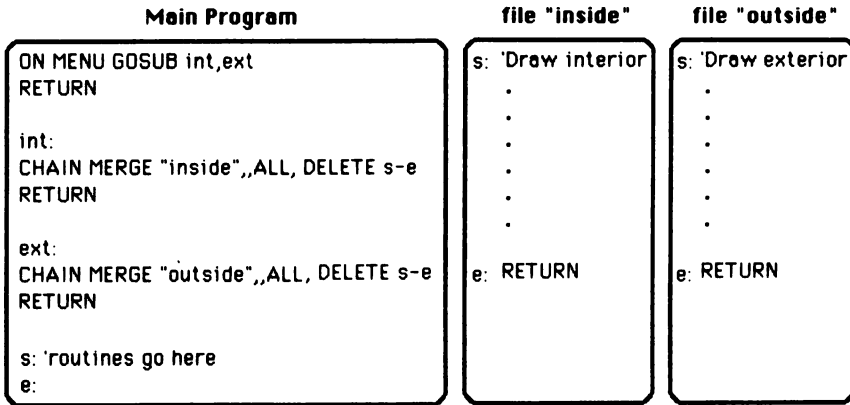


Figure 9-4.
Using CHAIN MERGE to overlay sections of a program

statement. You can then use the memory for other purposes or reuse it with the same array after redimensioning it:

```

GET (1,t)-(r,b),array
PUT (1+20,t),array,PSET
ERASE array
DIM array(30,12)
.
.
.

```

REPLACING CONSTANTS

Another way to save memory is to replace often-used constants (both numbers and strings) with variables. For example, a variable named *pi* consumes less memory than the constant 3.1415926. This is one technique that can actually improve the readability of your programs:

```

pi=3.1415927
FOR i=1 to 5
  FOR j=1 to 5 STEP .1
    x=200*cos(2*j*pi)*sin(i*pi/2)
    y=100*cos(2*j*pi)*cos(i*pi/2)
  NEXT j
NEXT i

```

Designing for Speed

In many graphics applications, the primary consideration is not how short you can make your program, but how fast you can display the graphics on the screen. Display speed is a function of both program design and the language used. The best language for fast program execution is the machine code of the computer's central processing unit (CPU). Programs written in machine code can take full advantage of the inherent speed of the computer. Another option is to write the majority of your program in BASIC, but code the speed-intensive routines in machine code and call them from the BASIC program. This technique uses both languages to their best advantage.

For many BASIC programmers, however, neither option is attractive. Machine code is difficult to learn, and programming with it requires heroic levels of concentration and patience. BASIC, on the other hand, is easy to learn, and Microsoft BASIC allows access to many of the routines in ROM that execute at machine-language speed.

Even so, BASIC programmers are handicapped by the fact that Microsoft BASIC on the Macintosh is an interpreted language; that is, each BASIC statement or function must be translated (interpreted) into 68000 machine code before it can be executed. This translation process is time-consuming, and it slows program execution considerably. Interpreted programs typically run ten times slower than their machine-language counterparts.

Fortunately, there are several ways to improve program execution speed. For example, consider that programs typically spend a good portion of their execution time processing a small portion of the code. You can speed execution by coding these high-use areas carefully.

CONDENSING CODE

Condensing the program code, as shown earlier in this chapter, not only saves memory but can also make your programs run faster. This means removing remarks and shortening variable and label names, removing all labels and line numbers not used by GOTOs and GOSUBs, and using multiple statements per line whenever possible. These changes will make your program less readable, but they will execute faster because the interpreter has less to read.

CONTROLLING VARIABLE TYPES

Another way to speed up a program is to choose variable types carefully. The default variable type for the decimal version of BASIC is

double-precision, and the default for the binary version is single-precision. You can change these defaults either by defining variable types with DEFINT, DEFSNG, DEFDBL, and DEFSTR statements, or by explicitly specifying each variable type with the symbols %, !, #, and \$, respectively. Defining variable types with DEF statements eliminates the need for these extra symbols and thus shortens variable names:

```
DEFINT a-l: DEFSNG m-y: DEFSTR z
FOR i=1 to 5
  READ e(i)
  DATA 3,5,2,7,5
NEXT i
p=3.1416: t=sin(p/2)
INPUT "Enter your name";znm
```

Of the three numeric variable types, BASIC processes integer variables most quickly because the Macintosh CPU can directly move and perform arithmetic on integers. The CPU requires more instructions to work with single- or double-precision variables. Remember, integer variables require only two bytes of storage, as compared to four bytes for single-precision and eight bytes for double-precision. The program simply has less data to manipulate with integer variables. Thus, you can speed up your programs considerably by using integer variables for loop counters, flags, and anything else in which decimal accuracy is not required.

HANDLING SUBROUTINES

Subroutines are very useful programming devices. They can cut down on program size by minimizing duplicated code. You can also use them to break large tasks into smaller units and organize programs logically, as in structured languages like Pascal. Unfortunately, heavy use of subroutines tends to slow down program execution. When calling a subroutine, the BASIC interpreter must save the current program location and find the destination label. Then it must restore the original program location at the end of the subroutine. All this overhead takes extra time.

You can eliminate jumps to short subroutines by replacing the GOSUB statement with a copy of the subroutine code. Figure 9-5 illustrates this technique.

CONTROLLING VARIABLE ORGANIZATION

The order in which you define and use string and array variables can be extremely important when you are using large arrays. If you ha-

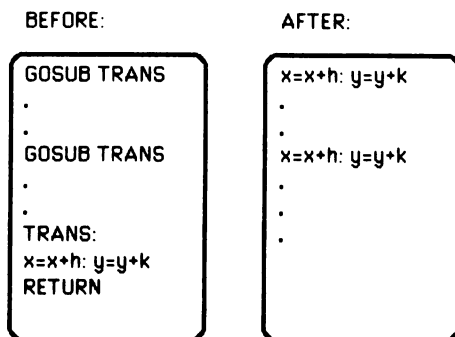


Figure 9-5.
Replacing GOSUB statements with subroutine code

bitually define your arrays before your string variables, you are asking for trouble. BASIC likes to store the names for all string variables in low memory before the array variables. Each time a program uses a new string variable, BASIC moves all the array variables higher in memory to make room for the new string. This is not meant to be devious; it is simply the way BASIC handles its variable storage. If you have a 512K or larger Macintosh, try this program:

```

DIM m$(30000)
PRINT "Variable Relocation Demonstration"
a$="aaaa": PRINT a$
b$="bbbb": PRINT b$
c$="cccc": PRINT c$
PRINT a$;b$;c$
a: IF INKEY$="" THEN a

```

The delay between print lines results from BASIC moving the array. Imagine how surprised an unsuspecting programmer would be to see his or her program suddenly pause right in the middle of a carefully choreographed graphics routine.

Now that you know the cause, what is the cure? Just mention each of the string variables to be used in your program before you dimension any arrays. Add this line to the beginning of the previous program:

```
a$="": b$="": c$=""
```

Note that the effect is more pronounced on programs with large amounts of data, particularly on Macintoshes with more than 128K of memory. As long as your program declares string variable names before dimensioning arrays, BASIC need not move the arrays.

You may also experience unexpected program delays when the computer accesses the disk to bring in new segments of the BASIC interpreter. Minimize this delay by allocating more heap space, so that more of BASIC is resident in memory (or increase your Macintosh's memory to 512K or more).

Speed Versus Memory

Execution speed and efficient use of memory are often conflicting factors in program design. In most cases, a program designed to execute quickly requires lots of memory. For example, to produce fast animation in BASIC, you need to prepare frames in advance, load them all into memory, and then display them rapidly by using PUT or PICTURE. Storing all these frames can put significant demands on memory.

If you change the program orientation to conserve memory, speed is likely to suffer. In the above example, you could store the frames on disk, load them one at a time into the same area of memory, and then display them on the screen. This requires less memory, but the transfer rate from disk to memory limits the display rate.

Summary

In this chapter, we covered several tips on how to get programs to run with limited memory and how to improve program speed. Don't try to use all these techniques in every program. Let necessity be your guide. You'll soon learn which methods work well with your programming style and which to use as a last resort.

Statements

CHAIN
CLEAR
ERASE

Function

FRE

Trademarks

The following italicized names are trademarked products of the corresponding companies.

<i>The Finder</i>	Apple Computer, Inc.
<i>MacPaint</i>	Apple Computer, Inc.
<i>MacDraw</i>	Apple Computer, Inc.
<i>MacWrite</i>	Apple Computer, Inc.
<i>Chart</i>	Microsoft Corporation
<i>MacVison</i>	Koala Technologies, Inc.
<i>Animation Toolkit 1</i>	Ann Arbor Softwork, Inc.
<i>Paint Mover</i>	Macinsoft, Inc.

Index

- ANIMATE command, 217
- Animation
 - multiple-frame, 203-12
 - on a background pattern, 203
 - program examples, 212-21
 - rotation, 208-09
 - single-frame, 200-03
- ARC call, 58, 71, 74
- Arcs, 64-66
- Arrays
 - defined, 22
 - to store images, 168-69
 - to store points, 23-24
 - transferring from a screen, 169-70
 - transferring from MacPaint, 218-21
 - transferring to a screen, 170-71
 - using to reclaim memory, 264-65
- BACKPAT statement, 47-48, 79
- BACKSPACE command, 9-10
- Bar charts, 81-83
- BASIC, uses in graphics, 2-8
- BEEP statement, 142-43, 144, 145, 150
- Beeps, 142-43
- BREAK call, 107
- BUTTON statement, 114
- Buttons, 114-15
- CALL statement, 36, 263
- Cartesian coordinate system, 37-38
- CHAIN MERGE statement, 264
- CHAIN statement, 263-64
- CHR\$ function, 30
- CIRCLE statement, 60, 63, 64, 68, 71, 85
- CIRCLE STEP statement, 61

- Circles, 60-63
- CLEAR statement, 260-61
- CLIP:PICTURE command, 177
- CLS (clear screen) statement, 17, 48
- COMMAND-, 75
- COMMAND-R, 52
- Commands
 - cursor, 105-07
 - draw, 173-74
 - print, 8-9, 10-11, 14-15, 181-85
- Computer-aided design (CAD)
 - program, 245-56
- COS statement, 40-41, 155, 229
- Cubes, 83-85
- Cursor
 - commands, 105-07
 - defining, 100-01
 - mask, 101-05
 - positioning, 107
- DATA statement, 47
- DEFINT statement, 25
- Dialog box, 113-14
- DIALOG call, 107, 112, 114-15
- DIM statement, 264
- EDITFIELD statement, 114
- ERASE statement, 68, 70, 76, 265
- Event trapping, 107-08
- FILES\$ function, 179
- FILL call, 76, 79, 83
- Filling in shapes, 68-80
- Fonts
 - described, 26-28
 - transferring, 31-33
- FOR/NEXT loop, 21, 22, 60
- FRAME call, 58, 60
- FRE function, 261
- Function plots, 38-40, 86-87
- GET statement, 15, 168, 169-70, 171, 176
- GETPEN function, 112
- HIDECURSOR call, 106
- HIDEPEN command, 173-74
- Image manipulation examples, 185-96
- Image storage
 - comparison, 174-76
 - in arrays, 168-71, 180-81
 - in strings, 171-73, 179-80
 - on disk, 178-79, 205-08
- Image transfer, 176-78
- INITCURSOR call, 105-06
- INKEY\$ function, 30-31, 161-62, 208
- INPUT command, 112
- INSTR function, 153
- Interactive programming
 - examples, 115-34
- INVERT call, 75
- INVERTRECT statement, 98
- LINE statement, 52, 66-67, 85, 87
- Lines, 52-53
- LLIST command, 9
- LOAD FILE command, 217
- LOADFRAME statement, 217
- LOCATE statement, 30
- Logo creation, 41-42, 90-91
- MacPaint
 - FatBits, 186-87
 - transferring into array format, 218-21
- Memory
 - controlling, 260-61
 - design, 262-65
 - monitoring, 261
 - partitions, 260
- MENU call, 107, 108
- Menu creation, 108-12
- MENU RESET command, 110
- Microsoft BASIC
 - getting started, 9-11
 - numeric variables, 25
- MOD function, 23
- MOUSE function, 95-97, 99, 107
- Mouse usage, 95-99
- MOVETO statement, 30
- OBSCURECURSOR call, 106-07
- ON/GOSUB command, 108, 109
- ON TIMER statement, 148, 150, 157

- Output window
 - changing sizes, 15-17
 - described, 14
 - user interaction, 112-13
- OVAL call, 58, 70
- Ovals, 63-64
- PAINT call, 68, 71, 74, 76
- PENPAT call, 79
- PICTURE OFF statement, 172
- PICTURE ON statement, 172, 173, 176, 179
- PICTURE statement, 172-73
- PICTURE\$ statement, 172, 179
- Pixels
 - absolute positioning, 21
 - defined, 8, 14
 - relative positioning, 21
- Plotted points
 - color options, 20-21
 - controlling patterns, 24
 - examples, 18-19, 42-48, 85-86, 87-90
 - Macintosh coordinate system, 17-18
 - randomness, 20, 21-22
 - speeding, 25-26
 - storing, 22-24
- POINT function, 25
- Polar functions, 40
- POLY call, 58
- Polygons, 53-57
- PRESET statement, 21
- PRINT statement, 19
- PSET statement, 19, 20-21, 37, 40, 170, 200, 202
- PUT statement, 15, 168, 170-71, 173, 187, 202, 205
- QuickDraw graphics package, 8, 36
- RANDOMIZE TIMER statement, 20, 55
- Read-only memory (ROM)
 - stored statements, 36
 - using to fill shapes, 67-80
- RECT call, 58, 83
- Rectangles, 57
 - rounded, 57-60
- RESTORE function, 144, 165
- RETURN command, 9, 109
- ROUNDRECT call, 58, 60
- RND function, 20, 55
- SAVE FILE command, 217
- SAVE FRAME statement, 217
- SETCURSOR call, 100-01
- SHIFT-COMMAND-3, 186
- SHIFT-COMMAND-4, 8, 10, 11, 14, 181
- SHOWCURSOR call, 106-07
- SHOWPEN command, 173-74, 176
- SIN statement, 40-41, 151, 155, 229
- Sound
 - defined, 138-39
 - program examples, 156-65
 - terminology, 139-42
 - with graphics, 147-51
- SOUND RESUME statement, 142, 153
- SOUND statement, 142, 143-44, 145, 148
- SOUND WAIT statement, 142, 153
- Speed design
 - condensing code, 266
 - controlling variable organization, 267-69
 - controlling variable types, 266-67
 - handling subroutes, 267, 268
- Square wave tones, 143-45
 - defining your own, 154-55
- STEP option, 21
- String/array conversion, 192-95
- SUB statement, 228-29
- System requirements, viii
- Text
 - face, 26, 34
 - mode, 26, 34-36
 - position, 28-31
 - selection, 31-33
 - size, 33-34
 - spacing, 26-28
- TEXTFACE call, 34
- TEXTFONT call, 32-33
- TEXTMODE call, 34

TEXTSIZE call, 33-34

Three-dimensional objects

 defined, 237-40

 manipulating, 241-44

 storing, 240, 241-44

TIMER call, 107

Transformation

 fixed points, 230-37

 rotation, 227

 scaling, 226

 subprograms, 228-30

 three-dimensional, 241

 translations, 224-25

User dialog, 112-13

VARPTR call, 58, 70

Voices

 multiple, 152-54

 single, 145-47

WAVE statement, 142, 151,
 154-55

WINDOW OUTPUT

 command, 9, 182-84

WINDOW OUTPUT #
 statement, 14

XOR statement, 170-71,
 176, 200

MACINTOSH™ GRAPHICS AND SOUND

PROGRAMMING IN MICROSOFT® BASIC

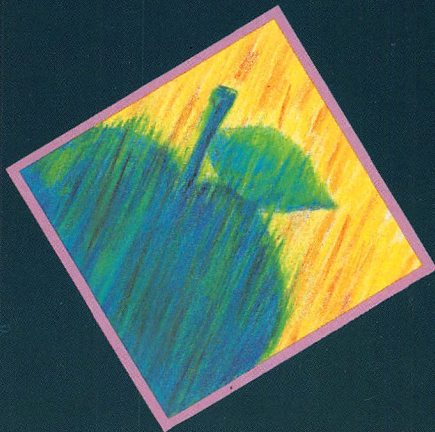
Create distinctive graphics, detailed three-dimensional drawings, exciting animation, and sound with your Macintosh.™

Whether you're a beginning or experienced computer user, **Macintosh™ Graphics and Sound** shows you how to program impressive graphics for entertainment, classroom, and business projects.

As you work with Kater's sample programs, you'll master all the Microsoft® BASIC (version 2.0) graphics commands as well as the Macintosh toolbox subroutines for cursor control, window management, and icon use.

From point plotting to figure transformations, **Macintosh™ Graphics and Sound** helps you build your programming skills to advanced level techniques for game design, presentation graphics, and more.

*Macintosh is a trademark of Apple Computer, Inc.
Microsoft is a registered trademark of Microsoft Corp.*



ISBN 0-07-881177-5